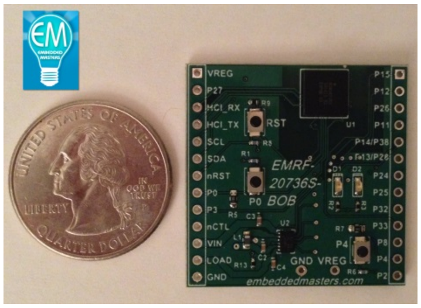
Embedded Masters

BCM20737S 1.3”x1.3” Breakout Board



Description:

The BCM20737S is one of the world’s smallest complete BTLE Modules at a mere 6.5x6.5mm. It comes in a 48pin LGA package that can go through a typical solder reflow process. It includes a complete BTLE Stack and a Cortex M3 running the show. User is able to make use of a FREE SDK(Software Development Kit) that is based on Eclipse and a GCC compiler. The SDK provides a simple Single-Click Installer. The SDK that supports the BCM20737S is SDK2.0.1 and beyond. The most current SDK is 2.1.0. This BOB is compatible with the examples provided in the SDK and makes use of the same pin allocations as used in the examples.

The EMRF-20737S provides an ultra-small Breakout Board that can either be used in your application or it can be used in place of the BCM92073x_LE_KIT commonly sold to support the BCM20736/BCM20737S. It comes with a TPS62740 ultra low Iq DC/DC converter with up to 90% efficiency even with light loads down to 10uA. The user has complete control over the output voltage of the DC/DC via Resistor Jumpers on the bottom of the board ranging from 1.8 to 3.3V in increments of 100mV. Default voltage is 3.0V. The TPS62740 also has a second voltage output(LOAD) for sensors and other miscellaneous devices. This is controlled via a CMOS GPIO pin(CTRL) which can be toggled HIGH/LOW to turn ON/OFF the secondary LOAD output. Every useable GPIO is routed out to 2.54mm headers. A few of the pins are connected to Pushbuttons or LEDs. The user has the capability of making use of every usable GPIO since they are routed out to the edge connectors. If the Pins that are connected to one of the Pushbuttons or LED’s is desired to be used the user can depopulate these components and make use of the GPIO pins with basic soldering skills. The only pin that is consumed is pin25/P1 which is internally connected to the EEPROM WP line and is connected to a pull-up resistor to allow for an always known state during RESET or POR events.

Gerber files and PCB Footprints are available upon request.
For more information on the BCM20737S interested users need to register on the Broadcom Community site at the following link to access the BCM20737S Technical Reference Manual, Appnotes, and associated information along with being able to download the SDK2.1.0 or Higher.
http://community.broadcom.com/welcome

Features:

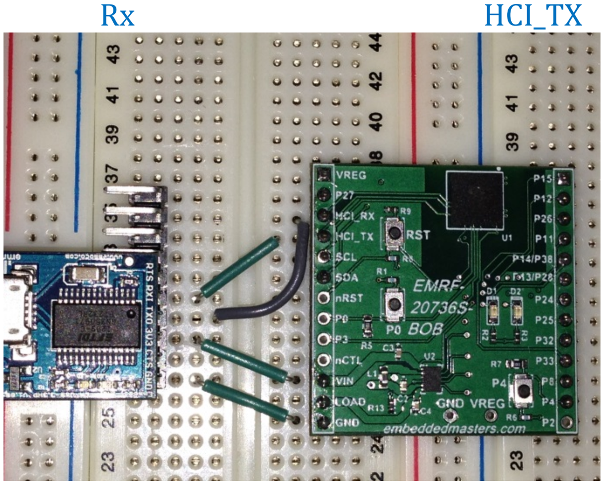
- VDD Supply voltage range of 1.62V–3.63V
- 1.3”x1.3” Breakout board that can be directly soldered into your application or be used for an ultra-small evaluation system.
- 90% Efficient at 10uA up to 300mA Output Current DC/DC Converter with Controllable Second LOAD output(TPS62740). 16 Selectable Output Voltage in 100mV steps from 1.8V to 3.3V.
- All Pins Broken Out to Standard 0.1"/2.54mm Spaced Headers

Section 1: Program the BCM20737S

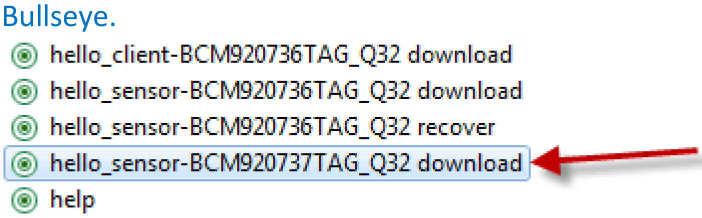
To program the BCM20737S on the Breakout Board you will need an external USB-UART/FTDI device to connect to the SDK provided by Broadcom. The BCM20737S accepts HCI3.0 commands from the SDK to program the EEPROM that is internal to the module. The BCM20737S only needs a simple 2 Wire UART to be programmed and the lines being HCI_RX and HCI_TX. Of course Power and GND are required also. The steps to program are outlined below.

1) Find your favorite USB-UART device. This is most likely going to be a FTDI 3.3V Cable, FT232/234 Breakout board, or you can actually even use an existing Broadcom BTLE TAG board if you happen to have one. Some common FTDI devices that can be used are outlined in the Appendix A.
   NOTE: Check back soon Embedded Masters will be making their own FT234 based Breakout similar to that shown in the picture below.
2) Connect the following lines of the FTDI device to the EMRF-20737S-BOB.
   NOTE: The BCM20737S IS NOT 5V tolerant!!

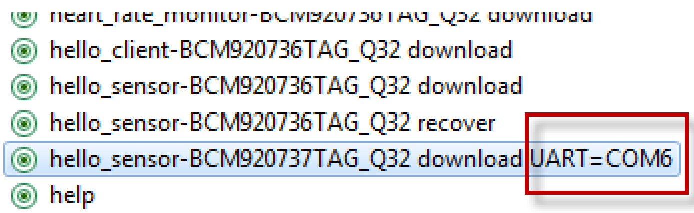
FTDI	EMRF-20737S-BOB
3.3(Power)	Vin(input into DC/DC regulator)
GND	GND
Tx	HCI_RX
Rx	HCI_TX

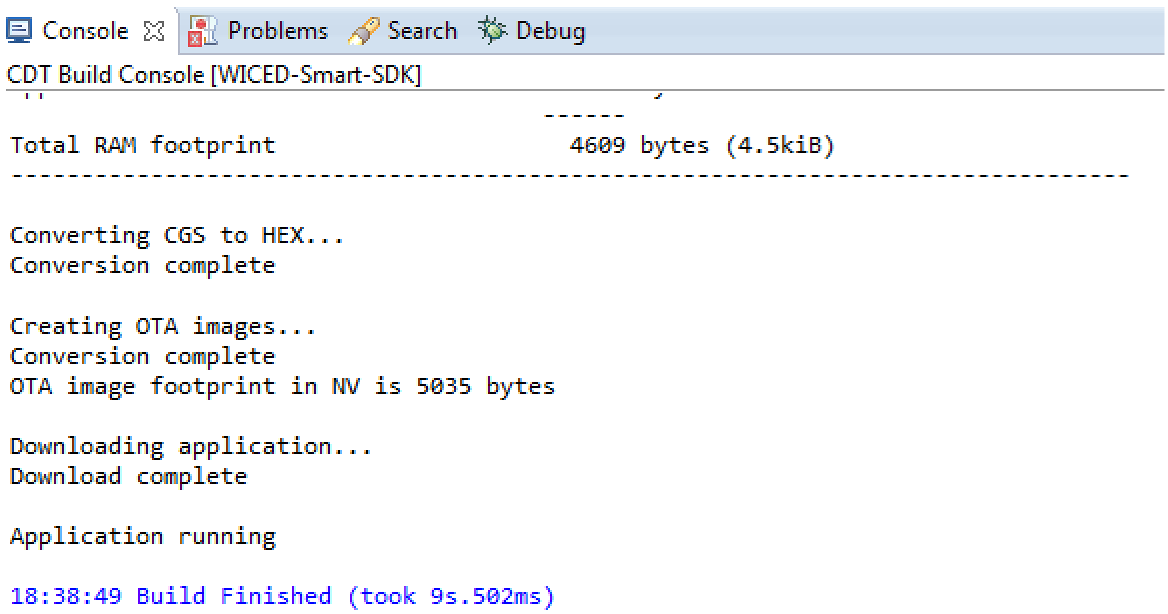


3) After you have done this you may want to ve very thorough and simply press the 'RST' button.
4) Now from the SDK simply click on your favorite Application Example 'Make Target' button as shown below. Let's try the ***hello_sensor*** App. To compile and download this App simply double-click on the Green Bullseye.



5) Now you should see in the Console Window that the App compiled, the Device was found, and the Download is Complete as shown below. Sometimes the 'Detecting Device…' process can take a while. You can speed this up by adding the COM port to the Make Target as shown below…





Section 2: Debug Output with the BCM20737S

In order to see the Debug Traces you need to simply disconnect the HCI_RX line from the FTDI TX line. So simply disconnect the 'wire' you have used to connect these two ports together. The HCI_RX line is important to the BCM2073x family as it is 'Sampled' upon every POR and RESET event by the BCM2073x and then determines whether the device should be in Programming Mode or in Application Mode. The Modes are indicated below…

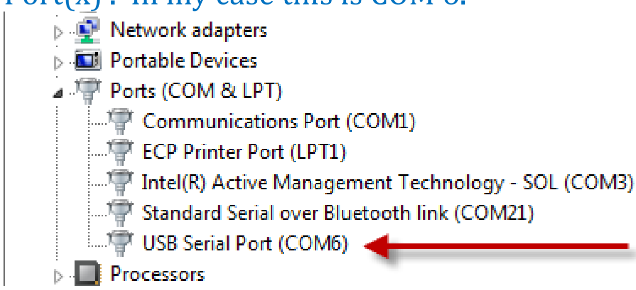
After a POR or RESET event:

HCI_RX = HIGH => Device enters Programming Mode. It may appear 'dead' if you have programmed it once and then hit RST on the BOB as it is waiting for HCI 3.0 programming commands.

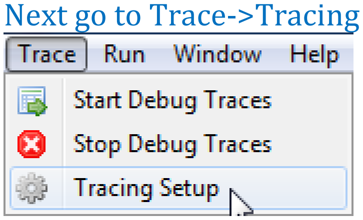
HCI_RX = LOW => Device enters Application Mode and begins executing program.

NOTE: After each time the device has been programmed it will begin executing afterwards but if you hit the RST button or Re-Apply Power it will go into either Programming Mode or Application Mode depending on whether HCI_RX is HIGH or LOW.

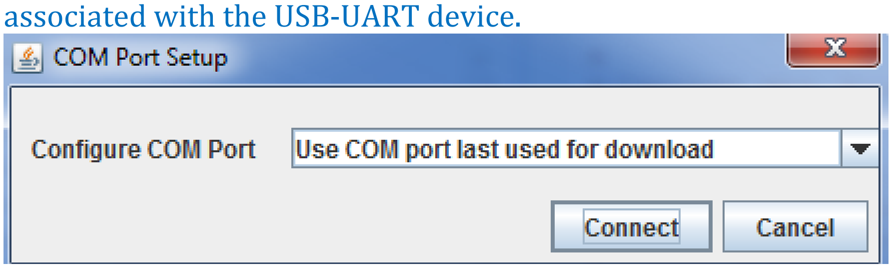
Now with HCI_RX explained we can get back to how to see the Debug Traces. With the HCI_RX line disconnected from FTDI_Tx line and you have hit RST on the BOB go to your 'Device Manager' or whatever tool will show you the COM ports attached to your system. Find the COM Port indicated as 'USB Serial Port(x)'. In my case this is COM 6.



Next go to Trace->Tracing Setup in SDK2.x.x.



If you are using SDK2.0.1 it will take some time and it may seem like the SDK is hung but it is not…you will eventually see the GUI below. SDK2.1.0 and beyond does this much faster. Choose your COM port that is associated with the USB-UART device.



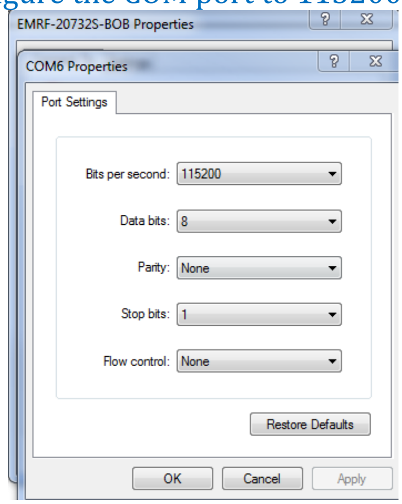
You will now see the following output in the Console Tab…

The initial values printed out are values that were loaded into the GATT database and also various BTLE stack configurations and GPIO settings that have been configured in the BTLE Stack. After that you will see the hello_sensor_timeout: x being printed out which is the 1 second timer.

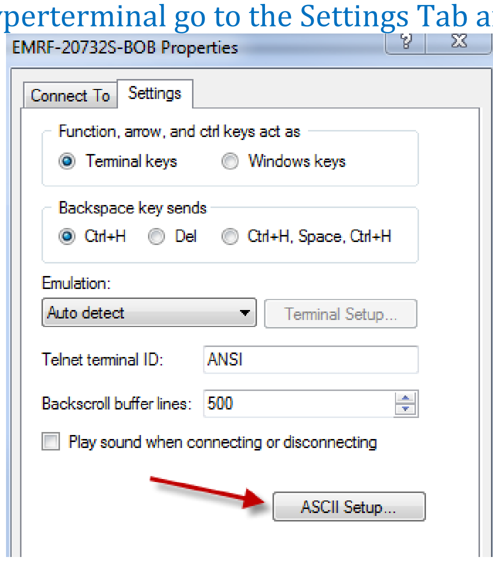
```
Console  Problems  Search  Debug
Broadcom Debug Viewer
00:40:26 -
00:40:26 - blecm evt handler:
00:40:26 - 0e04010a2000
00:40:26   Trace Decoding Error - Could not find line number 314
00:40:26   UUID  : 2800
00:40:26   Attribute bytes
00:40:26   Handle: 0062
00:40:26   Perm  : 0002
00:40:26   Len, Max Len : 0005, 0005
00:40:26   UUID  : 2803
00:40:26   Attribute bytes
00:40:26   Handle: 0063
00:40:26   Perm  : 0002
00:40:26   Len, Max Len : 0001, 0001
00:40:26   UUID  : 2A19
00:40:26   Attribute bytes
00:40:26   Gatt DB Dump complete
00:40:26   bd_addr[5:2] = 20 73 6A 18
00:40:26   bd_addr[1:0] = 9877 00
00:40:26   GPIO 0001 (11)
00:40:26   GPIO 0000 (104)
00:40:26   GPIO 0014 (1003)
00:40:26   GPIO 0015 (20)
00:40:26   GPIO 0028 (2001)
00:40:26   Interrupt mask[0,1]:0001 0000
00:40:26   Interrupt mask[2]:0000
00:40:26   GPIO_WP:OFF= 00
00:40:26   GPIOBTN1:OFF=1,INT:0
00:40:26   GPIO_LED:OFF=1
00:40:26   GPIOBAT
00:40:26   GPIO_BUZ:OFF=0
00:40:26   Battery level: 0/100
00:40:26   Fine Timer(0 ms, 0/sec)
00:40:26   Fine TImer tick 80
00:40:26   Normal Timer(1 s, 80 tick)
00:40:26   BLE_high_un_adv:timer(0)
00:40:26 - hello_sensor_timeout:0
00:40:27 -
```

You could choose to not use the Trace inside of the SDK2.x.x and use your own favorite terminal window but some of the data that will show on the terminal window will not have as much detail for the items that are printed from ROM. Regardless, to do so you would follow the instruction below. Any terminal window, Putty, RealTerm, TeraTerm will work and will have similar setup as HyperTerminal which is shown below.

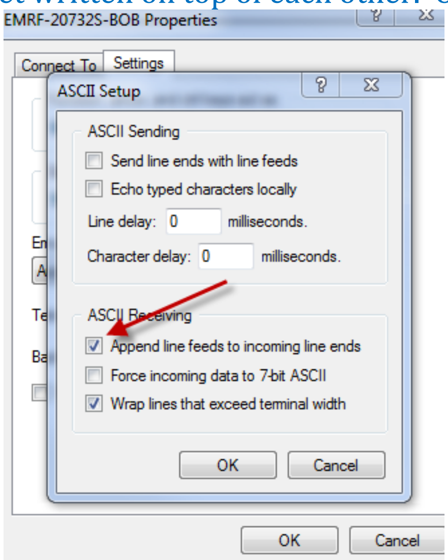
1) Configure the COM port to 115200, 8, N, 1, N as shown below…



2) In Hyperterminal go to the Settings Tab and press the 'ASII Setup…' button

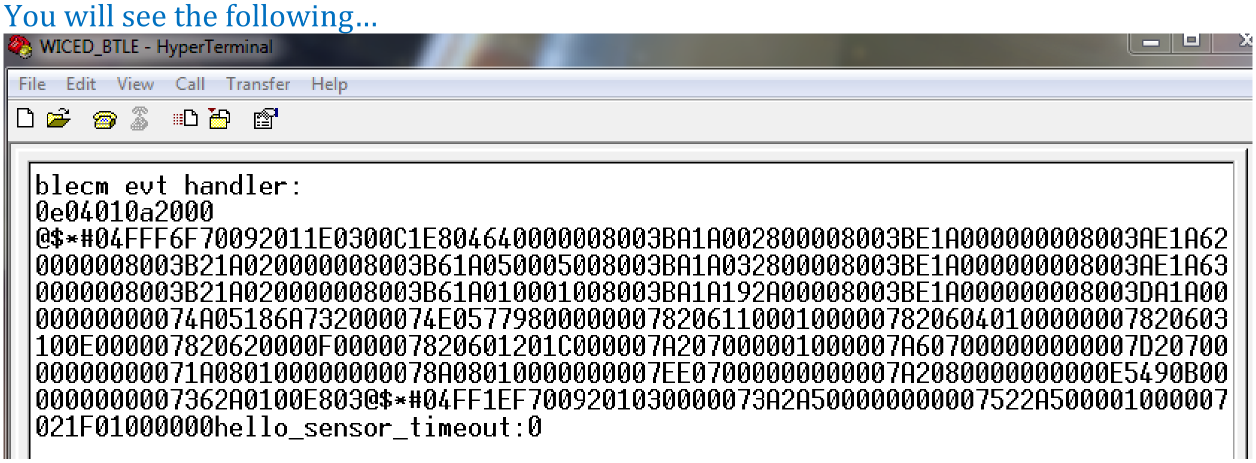


3) Then check the box next to ‘Append line feeds to incoming line ends’ as shown below. This will ensure that you see all the lines that are being printed out of the device. If this is not checked some of the ble_trace/debug statements in ROM do not have a line feed at the end of the print statement so they will get written on top of each other. Other terminal windows have a similar function to enable this.



4) Now click OK, OK to get out of the ASII Setup and Properties Dialogue and connect to the COM port. You will start seeing Debug Messages being printed out to the Terminal window as such. You may want to press ‘RST’ on the EMRF board to start from the beginning. Note you will see the printout from ROM as a bunch of numbers without any indication as to what it is. Regardless, any Debug messages you put in your firmware will print out correctly.

You will see the following...



You have now successfully programmed the BCM20737S and enabled the Debug output. Now let's touch on a few more details as outlined below.

Section 3. hello_sensor Walk Through
Section 4. hello_sensor Code Analysis
Section 5. How do I create my own project?
Section 6. Debugging Techniques.
Section 7. How To Sleep?
Section 8. How do I configure GPIO?

These topics will be covered in the next sections. Embedded Masters will be creating additional documents that will explain even further details of the BCM20732S, BCM20736S and BCM20737S modules. One of the really cool things about these modules is they are all 100% pin compatible!!

Section 3: hello_sensor Walk Through

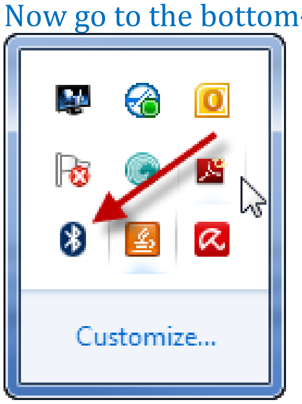
What does this App Do?

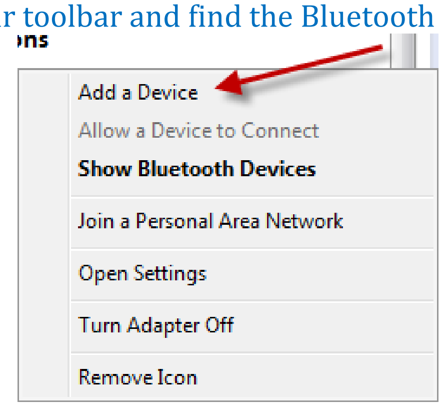
First to fully utilize the ***hello_sensor*** app you will want a PC that you can connect to a BTLE device. WIN8 already has this built in. If you have WIN7 or XP don't worry you can get a ~$13 USB dongle from Plugable and this will update your WIN7 or XP machine to have BTLE capability. The link to purchase this dongle and download the drivers for it is provided below.

http://plugable.com/products/usb-bt4le

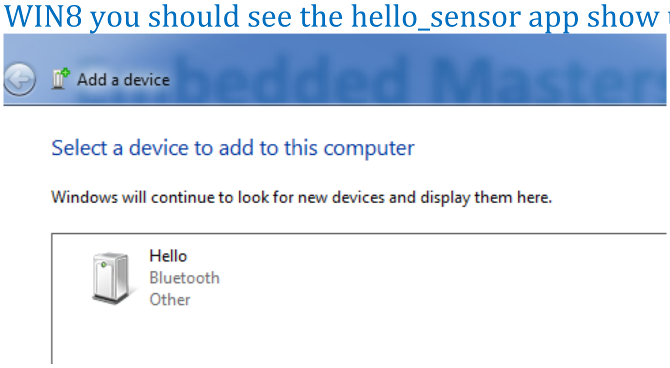
Once you have one of these and you have installed the drivers you will be ready to follow along. Assuming you have followed the steps in the Intro section and have programmed the **EMRF-20737S** with the ***hello_sensor*** app we will do the following steps to run the ***hello_sensor*** app.

1) Ensure you have disconnected the **HCI_RX** line from the **FTDI_TX** line so you can see the Debug output on a terminal window as shown in the prior section.
2) Now go to the bottom-right of your toolbar and find the Bluetooth symbol and select 'Add Device'



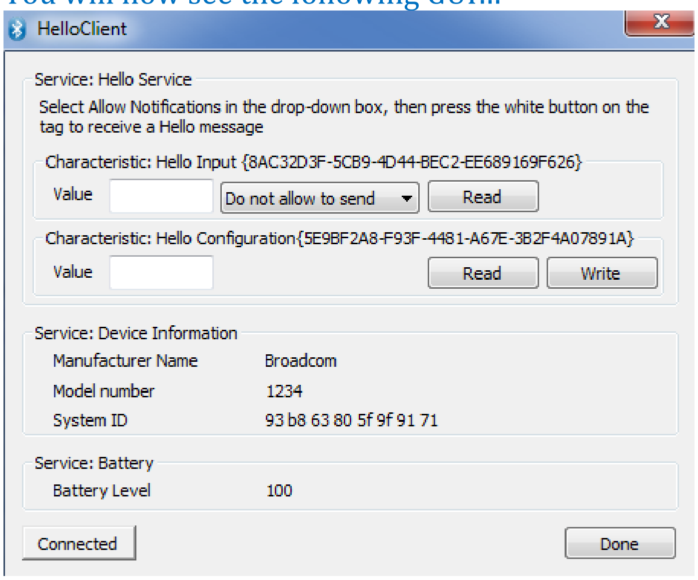


3) If your Plugable dongle and WIDCOM Bluetooth drivers have installed correctly OR you are using WIN8 you should see the hello_sensor app show up after selecting 'Add a Device'



4) After click on the ‘Hello’ symbol your system will attempt to go find drivers for ‘Hello’. There are no drivers for it so you could skip the driver installation/search if you want.
5) Now go back to SDK2.x.x and find in the ***hello_sensor*** folder the **Windows->Release** folder. Choose the appropriate folder: x64(64bit OS) or x86(32bit OS) and click on the ***HelloClient.exe***

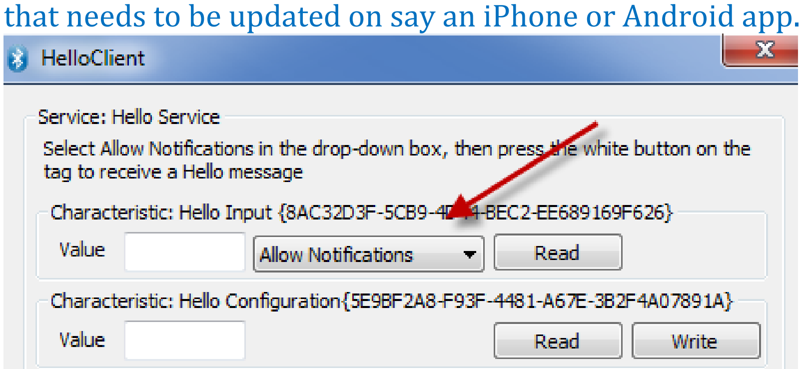
6) You will now see the following GUI…



7) The GUI allows communication from the BTLE dongle to the EMRF board. To make use of the GUI you can do the following:

A. <u>Indications/Notifications(CLIENT_CONFIGURATION_DESCRIPTOR)</u>

i. Select ‘Allow Notifications’ or ‘Allow Indications’ from the drop-down menu. Notifications/Indications are messages that are sent from the ***EMRF-20737S***(Slave) to the PC(Master). In a real application these could be updates from a sensor or some other data that needs to be updated on say an iPhone or Android app.

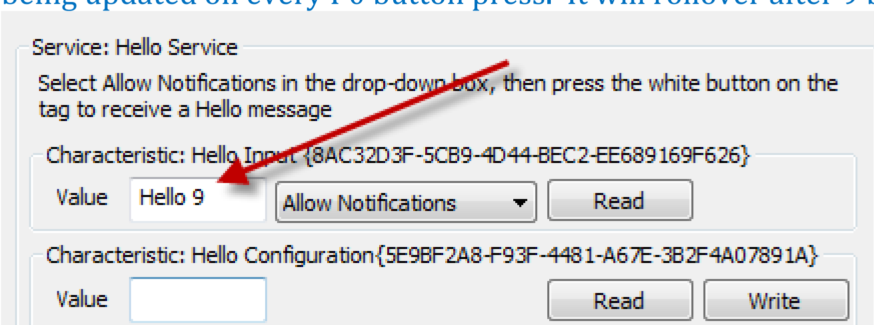


In the trace/terminal window you will see the following indicating this has occurred.

```
00:17:34 - l2cap Rx:
00:17:34 - 4020090005000400122b000100
00:17:34 -
00:17:34 - Checking readable attribute 002b
00:17:34 -
00:17:34 -  permission check retCode = 00
00:17:34 - hello_sensor_write_handler: handle 002b
00:17:34 -
00:17:34 - hello_sensor_write_handler: client_configuration 0001
00:17:34 -
00:17:34 - hello_sensor_write_handler: NVRAM write:0009
00:17:34 -
00:17:34 -
00:17:34 - l2cap Tx:
00:17:34 - 402005000100040013
00:17:34   WriteCb: handle 002b
00:17:34   BUZBeep(0)
00:17:35 -
00:17:35 - blecm evt handler:
00:17:35 - 13050140000100
00:17:35 - hello_sensor_timeout:154
```

ii. Now click the 'Read' button on the GUI. You will see Hello 0 being displayed.

iii. To send updates to the PC app click on the P0 Pushbutton. You will see the last digit/byte is being updated on every P0 button press. It will rollover after 9 back to 0.

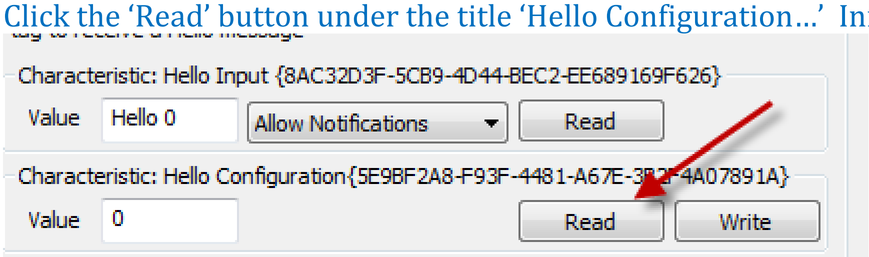


On every button press you will see something similar to below in the trace/terminal window indicating that an Interrupt has occurred.

```
00:19:22 -
00:19:22 - hello_sensor_timeout:264
00:19:22 -
00:19:22 - (INT)But1:1 But2:0 But3:0
00:19:22 -
00:19:22 - 48656c6c6f2030
00:19:22 -
00:19:22 -  permission check retCode = 00
00:19:22 - hello_sensor_send_message
00:19:22 -
00:19:22 - hello_sensor_indication_sent:  0000
00:19:22 -
00:19:22 - 48656c6c6f2031
00:19:22 - SENSOR_VALUE_NOTIFY db_pdu.len: 0007
00:19:22 -
00:19:22 -
00:19:22 - l2cap Tx:
00:19:22 - 40200e000a0004001b2a0048656c6c6f
00:19:22 - 2031
00:19:22 -
00:19:22 -
00:19:22 - hello_sensor_send_message->InterruptHandler
00:19:22 - (INT)But1:0 But2:0 But3:0
00:19:22 -
00:19:22 - 48656c6c6f2031
00:19:22 -
00:19:22 -  permission check retCode = 00
00:19:22 - hello_sensor_send_message
00:19:22 -
00:19:22 - hello_sensor_indication_sent:  0000
00:19:22 -
00:19:22 - 48656c6c6f2032
00:19:22 - SENSOR_VALUE_NOTIFY db_pdu.len: 0007
00:19:22 -
00:19:22 -
00:19:22 - l2cap Tx:
00:19:22 - 40200e000a0004001b2a0048656c6c6f
00:19:22 - 2032
00:19:22 -
00:19:22 -
00:19:22 - hello_sensor_send_message->InterruptHandler
00:19:22 -
00:19:22 - blecm evt handler:
00:19:22 - 13050140000100
00:19:22 -
00:19:22 - blecm evt handler:
00:19:22 - 13050140000100
00:19:23 - hello_sensor_timeout:265
```

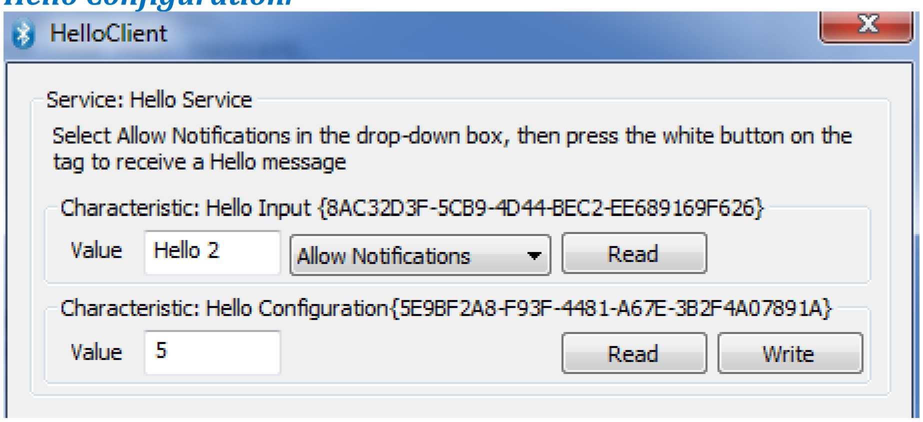
B. UUID_HELLO_SENSOR_CONFIGURATION

i. Click the ‘Read’ button under the title ‘Hello Configuration…’ Initially you will see 0.



ii. Now type in a value such as 5 and click ‘Write’. You will see D1 blink 5 times.

iii. Now if you hit P0 again you will not only see the last digit increment in Hello X but you will also see the LED flash the same number of times that you wrote into the ***Characteristic: Hello Configuration.***



We are now done with the hello_sensor App walk-through. Next we will do a code analysis for the hello_sensor app.

Section 4: hello_sensor code analysis.

Now that we have seen what the ***hello_sensor*** app does let's inspect the code to see how it all goes together.

1) GATT Database Configuration

The GATT(Generic Attribute Profile) database configures all the 'services' and 'characteristics' for a given BTLE application. You can think of services as functions that handle particular data types and provide various functionality. Many services are part of the BTv4.0/v4.1 spec and help comprise BTLE profiles. Characteristics can be thought of as a description of the data variable/value that is used in the service. The GATT database is read by a client/master during the connection process so that it understands the services that are offered by the slave/server. The definitions in the GATT database effectively provides a specification as to how the devices should pass data and/or communicate. This is a rather generic description of the GATT database and characteristics it is a bit more sophisticated than that but this will serve the purpose of the discussion here. For further reading I would suggest actually downloading the Bluetooth Core Doc Specification manual and read through the BTLE sections. Don't get overly concerned that the Core Doc PDF is 2300+ pages there are only 3-4 sections that are vital for BTLE. Minimally I would recommend reading the BTLE sections in Volume 1 of the Core Docs which will provide a good overview of BTLE and how it is structured.

Looking at the GATT database for hello_sensor we see the following…

```
/*
* This is the GATT database for the Hello Sensor application.  It defines
* services, characteristics and descriptors supported by the sensor.  Each
* attribute in the database has a handle, (characteristic has two, one for
* characteristic itself, another for the value).  The handles are used by
* the peer to access attributes, and can be used locally by application for
* example to retrieve data written by the peer.  Definition of characteristics
* and descriptors has GATT Properties (read, write, notify...) but also has
* permissions which identify if application is allowed to read or write
* into it.  Handles do not need to be sequential, but need to be in order.*/
const UINT8 hello_sensor_gatt_database[]=
{
// Handle 0x01: GATT service
// Service change characteristic is optional and is not present
```

So we can see the GATT database is essentially an array of data that defines the Services, Characteristics, and Descriptors. A couple of keys to keep in mind when reading through this section of firmware are the following…

1) Handles are simply addresses that make it handy to reference the individual elements with.
   NOTE: Handles do not have to be consecutive but MUST be in order.
2) Services, Characteristics, etc defined with a UUID16 are all assigned by the Bluetooth Sig.
3) Services, Characteristics, etc defined with a UUID128 are all custom definitions that are specific to the application.

What is ***UUID_SERVICE_GATT***? If we do a search in the SDK we find that there is a file ***ble_uuid.h*** in which the BT Sig defined UUID Services are defined. If we look in that file we see that ***UUID_SERVICE_GATT*** is defined as 0x1801. We can double check this by looking at the Bluetooth.org site and we see exactly that. The GAP Service further down shows as 0x1800 in ***ble_uuid.h*** which is matches exactly what the BT Sig definition is. I might recommend to review the other services available that are defined in this file.
https://www.bluetooth.org/en-us/specification/assigned-numbers/generic-attribute-profile

GATT Services

Mnemonic	UUID Size	UUID	Referenced Specification
<<Generic Access Profile>>	uuid16	0x1800	*Bluetooth*® Core Specification Volume 3, Part C, Section 12
<<Generic Attribute Profile>>	uuid16	0x1801	Bluetooth Core Specification Volume 3, Part G, Section 7

```
PRIMARY_SERVICE_UUID16 (0x0001, UUID_SERVICE_GATT),
// Handle 0x14: GAP service
// Device Name and Appearance are mandatory characteristics.  Peripheral
// Privacy Flag only required if privacy feature is supported.  Reconnection
// Address is optional and only when privacy feature is supported.
// Peripheral Preferred Connection Parameters characteristic is optional
// and not present.
PRIMARY_SERVICE_UUID16 (0x0014, UUID_SERVICE_GAP),
// Handle 0x15: characteristic Device Name, handle 0x16 characteristic value.
// Any 16 byte string can be used to identify the sensor.  Just need to
// replace the "Hello" string below.  Keep it short so that it fits in
// advertisement data along with 16 byte UUID.
```

Now we are going to define some characteristics such as what the 'Device Name' is. The device name is what will show up after a Master has read the GATT database and has established a connection. This 'Characteristic' is a BT Defined value in which the UUID is again defined in ***ble_uuid.h***. Note that there are 2 handle values as indicated in the comments 0x0015 for the Device name and 0x0016 for the 'characteristic' value. Also note the 'characteristic' has been defined as a readable value and the value can be up to 16bytes long. If you wanted you could give your device a custom name by replacing the *Hello* with your own name that can be up to 16bytes long. **NOTE:** If you want to look at how the ***CHARACTERISTIC_UUID16()*** function prototype looks you find the function definition in ***bleprofile.h***

```
CHARACTERISTIC_UUID16 (0x0015, 0x0016, UUID_CHARACTERISTIC_DEVICE_NAME,
                              LEGATTDB_CHAR_PROP_READ, LEGATTDB_PERM_READABLE, 16),
 'H','e','l','l','o',0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
```

This characteristic tells the device it is connecting to what kind of device it is. A PC for example may show different symbols based on what the APPEARANCE is defined as. There is a list of values for APPEARANCES in ***ble_uuid.h***. **NOTE:** We have 2 handles defined here. One for the ***UUID_CHARACTERISTIC_APPEARANCE*** and one for the Value of the APPEARANCE being ***APPEARANCE_GENERIC_TAG***.

```
// Handle 0x17: characteristic Appearance, handle 0x18 characteristic value.
// List of approved appearances is available at bluetooth.org.  Current
// value is set to 0x200 - Generic Tag
CHARACTERISTIC_UUID16 (0x0017, 0x0018, UUID_CHARACTERISTIC_APPEARANCE,
                              LEGATTDB_CHAR_PROP_READ, LEGATTDB_PERM_READABLE, 2),
  BIT16_TO_8(APPEARANCE_GENERIC_TAG),
```

Now we define the actual 'custom' service which is specified by a UUID128 since it is NOT a BT Sig defined service. The Handle and Service are defined in hello_sensor.h. You may be asking how does one define this 128bit UUID? Well there is not really a defined way to do this. There are some pretty slick online applications that can do this for you such as the one from the link below…

http://www.guidgenerator.com/

```
// Handle 0x28: Hello Service.
// This is the main proprietary service of Hello Sensor.  It has 2 characteristics.
// One will be used to send notification(s) to the paired client when button is
// pushed, another is a configuration of the device.  The only thing which
// can be configured is number of times to send notification.  Note that
// UUID of the vendor specific service is 16 bytes, unlike standard Bluetooth
// UUIDs which are 2 bytes.  _UUID128 version of the macro should be used.
PRIMARY_SERVICE_UUID128 (HANDLE_HELLO_SENSOR_SERVICE_UUID, UUID_HELLO_SERVICE),
```

Now we are going to define a UUID128 characteristic that will be passed during a Notification or an Indication. Note that the Characteristic is Readable and is a 7byte value. We will see later how this value gets updated in the application the button press. The difference between a Notification and an Indication is that an Indication requires a response from the client/master as confirmation that it has received the message. In this App the user selects whether it is a Notification or Indication via the PC GUI drop down menu. Indications/Notifications are quite useful as they allow the slave/server to update the master/client when a new measurement has been taken or the value has been updated somehow versus having to have the master/client continually poll the slave.

```
// Handle 0x29: characteristic Hello Notification, handle 0x2a characteristic value
// we support both notification and indication.  Peer need to allow notifications
// or indications by writing in the Characteristic Client Configuration Descriptor
// (see handle 2b below).  Note that UUID of the vendor specific characteristic is
// 16 bytes, unlike standard Bluetooth UUIDs which are 2 bytes.  _UUID128 version
// of the macro should be used.
CHARACTERISTIC_UUID128 (0x0029, HANDLE_HELLO_SENSOR_VALUE_NOTIFY,
       UUID_HELLO_CHARACTERISTIC_NOTIFY,LEGATTDB_CHAR_PROP_READ|LEGATTDB_CHAR_PROP_NOTIFY|
       LEGATTDB_CHAR_PROP_INDICATE, LEGATTDB_PERM_READABLE, 7),
       'H','e','l','l','o',' ','0',
```

We are now creating a ***Descriptor*** for the client/master to indicate whether it shall receive Indications or Notifications. Note this is defined as a 2byte value as described in the comments below.

https://developer.bluetooth.org/gatt/descriptors/Pages/DescriptorViewer.aspx?u=org.bluetooth.descriptor.gatt.client_characteristic_configuration.xml

```
// Handle 0x2b: Characteristic Client Configuration Descriptor.
// This is standard GATT characteristic descriptor.  2 byte value 0 means that
// message to the client is disabled.  Peer can write value 1 or 2 to enable
// notifications or indications respectively.  Not _WRITABLE in the macro.  This
// means that attribute can be written by the peer.
CHAR_DESCRIPTOR_UUID16_WRITABLE (HANDLE_HELLO_SENSOR_CLIENT_CONFIGURATION_DESCRIPTOR,
        UUID_DESCRIPTOR_CLIENT_CHARACTERISTIC_CONFIGURATION,
        LEGATTDB_PERM_READABLE | LEGATTDB_PERM_WRITE_REQ, 2),
  0x00,0x00,
```

Here we are simply defining a ***Characteristic*** that stores the value for how many ***Notifications*** or ***Indications*** will be sent after a button press. **NOTE:** it is a single byte.

```
// Handle 0x2c: characteristic Hello Configuration, handle 0x2d characteristic value
// The configuration consists of 1 bytes which indicates how many notifications or
// indications to send when user pushes the button.
CHARACTERISTIC_UUID128_WRITABLE (0x002c, HANDLE_HELLO_SENSOR_CONFIGURATION,
       UUID_HELLO_CHARACTERISTIC_CONFIG,LEGATTDB_CHAR_PROP_READ | LEGATTDB_CHAR_PROP_WRITE,
       LEGATTDB_PERM_READABLE | LEGATTDB_PERM_WRITE_CMD | LEGATTDB_PERM_WRITE_REQ,  1),
  0x00,
```

Here we are defining a standard BT defined service which provides the ***Device Information***.

```
// Handle 0x4d: Device Info service
// Device Information service helps peer to identify manufacture or vendor
// of the device.  It is required for some types of the devices (for example HID,
// and medical, and optional for others.  There are a bunch of characteristics
// available, out of which Hello Sensor implements 3.
PRIMARY_SERVICE_UUID16 (0x004d, UUID_SERVICE_DEVICE_INFORMATION),
```

Here we are simply defining the ***Characteristic*** of the ***Device Information*** in which we providing a manufacturer name being Broadcom. **NOTE:** This is where you would define your own company name. Ensure you define it with enough bytes to completely store the name you want.

```
// Handle 0x4e: characteristic Manufacturer Name, handle 0x4f characteristic value
CHARACTERISTIC_UUID16 (0x004e, 0x004f, UUID_CHARACTERISTIC_MANUFACTURER_NAME_STRING,
       LEGATTDB_CHAR_PROP_READ, LEGATTDB_PERM_READABLE, 8),
  'B','r','o','a','d','c','o','m',
```

Here we are simply using another BT defined value which is the Model number. For your own product you would provide a rev #, product ID, or something that indicates what the product/model is.

```
// Handle 0x50: characteristic Model Number, handle 0x51 characteristic value
CHARACTERISTIC_UUID16 (0x0050, 0x0051, UUID_CHARACTERISTIC_MODEL_NUMBER_STRING,
       LEGATTDB_CHAR_PROP_READ, LEGATTDB_PERM_READABLE, 8),
  '1','2','3','4',0x00,0x00,0x00,0x00,
```

Here we are defining the ***SYSTEM_ID***. The ***SYSTEM_ID*** is a BT Sig defined value which is a 64bit value. The 64bit value is broken into a 40bit manufacturer-defined identifier concatenated with a 24bit unique Organizationally Unique Identifier(OUI). The OUI is issued by the IEEE Registration Authority and is required to be used in accordance with IEEE Standard 802-2001.6 while the last 40bits Are manufacturer defined. http://standards.ieee.org/regauth/index.html

```
// Handle 0x52: characteristic System ID, handle 0x53 characteristic value
CHARACTERISTIC_UUID16 (0x0052, 0x0053, UUID_CHARACTERISTIC_SYSTEM_ID,
       LEGATTDB_CHAR_PROP_READ, LEGATTDB_PERM_READABLE, 8),
  0x93,0xb8,0x63,0x80,0x5f,0x9f,0x91,0x71,
```

Here we are defining a BTLE Standard service which is to monitor the battery level.

```
// Handle 0x61: Battery service
// This is an optional service which allows peer to read current battery level.
PRIMARY_SERVICE_UUID16 (0x0061, UUID_SERVICE_BATTERY),
```

Here we are simply defining the 'Characteristics' of the Battery Service itself. **NOTE:** It is 1 byte.

```
// Handle 0x62: characteristic Battery Level, handle 0x63 characteristic value
CHARACTERISTIC_UUID16 (0x0062, 0x0063, UUID_CHARACTERISTIC_BATTERY_LEVEL,
                       LEGATTDB_CHAR_PROP_READ, LEGATTDB_PERM_READABLE, 1),
  0x64,
};
```

2) BLE_PROFILE_CFG: Stack Configuration

This structure defines how the Broadcom BTLE Stack gets initialized. We will go through the most important items here.

```
const BLE_PROFILE_CFG hello_sensor_cfg =
{
```

Here we define the fine_timer_interval. Think of this as your fast application system tick

```
/*.fine_timer_interval            =*/ 250,  // ms
```

Here we are simply defining that the Advertising be ***UNDIRECTED_DISCOVERABLE***. This simply means that any BTLE device can see the advertisements and 'Discover' or connect to it. There are several types of Advertisements that one can use. For example if you know the Device you want to connect to you might use a ***DIRECTED_DISCOVERABLE*** advertisement which means it is specific to a particular BTLE device and uses that devices BD(Bluetooth Device) Address. If you are designing a Beacon type of product you would make it so it is NOT Discoverable meaning a device is not allowed to connect to it as it is simply intended to send one way information.

```
/*.default_adv                    =*/ 4,    // HIGH_UNDIRECTED_DISCOVERABLE
```

We won't worry about this setting, just leave as 0.

```
/*.button_adv_toggle              =*/ 0,    // pairing button make adv toggle (if 1)
                                            // or always on (if 0)
```

We now define the ***High and Low Advertising Intervals***. Keep in mind that the Advertising Interval is strictly defined by the BT Sig to be a multiple of .625ms. So for the High Advertising Interval this is defined as 32 * .625ms = 20ms intervals. The Low Advertising interval is less frequent and is 1024 * .625ms = 640ms. We then define the High Adv duration to be 30 seconds and the Low to be 300 seconds long. The Low and High intervals are used because typically an BTLE device has a use case when it is user directed say by an On/Off button press and the system knows it should be in a scenario where the user is trying to connect the device to BTLE client. Keep in mind ONLY slaves/servers advertise it is the master/clients that see the advertisements. Having a High and Low allows the system to send advertising packets at a fast interval to try to establish the connection quickly. If a connection is not made within the specified High Adv Duration then it goes into a lower power mode and advertises at a slower rate. This is entirely user configurable and typically the user defines a callback function to determine what to do if after the High and Low Advertising durations have expired.

```
/*.high_undirect_adv_interval     =*/ 32,   // slots
/*.low_undirect_adv_interval      =*/ 1024, // slots
/*.high_undirect_adv_duration     =*/ 30,   // seconds
/*.low_undirect_adv_duration      =*/ 300,  // Seconds
```

```
/*.high_direct_adv_interval       =*/ 0,    // seconds
/*.low_direct_adv_interval        =*/ 0,    // seconds
/*.high_direct_adv_duration       =*/ 0,    // seconds
/*.low_direct_adv_duration        =*/ 0,    // seconds
```

We now define the ***local name***. This should be the same as the ***Device Name*** that is specified in the GATT database. You could actually define them as different names. What would happen is that the local name below is what would get advertised and once the client/master sees the advertisements and reads the GATT database it will pull out the Device Name specified in the GATT database.

```
/*.local_name                     =*/ "Hello",         // [LOCAL_NAME_LEN_MAX];
/*.cod                            =*/ "\x00\x00\x00", // [COD_LEN];
/*.ver                            =*/ "1.00",          // [VERSION_LEN];
```

Here we are indicating that we are requiring that the device create a 'Secure' connection.

```
/*.encr_required                  =*/ (SECURITY_ENABLED | SECURITY_REQUEST),    // data
                                     // encrypted and device sends security request
                                     // on every connection
/*.disc_required                  =*/ 0,    // if 1, disconnection after confirmation

/*.test_enable                    =*/ 1,    // TEST MODE is enabled when 1
```

Here is where we specify the output PA level. It can be as high as +4dBm.

```
/*.tx_power_level                 =*/ 0x00, // dbm
```

Here is where you could specify a time in seconds that if the connection is IDLE that the connection times out and disconnects. For development purposes it is easier to just set this to 0 so it never times out.

```
/*.con_idle_timeout               =*/ 0,    // second  0-> no timeout
```

Here is where you can specify that if the system is IDLE for a time period(in seconds) that the device goes into a low power mode. Personally I have found it is better to do this manually in the firmware. We will cover how to do this in **Section 7** and you will see firmware that allows this later in this section.

```
/*.powersave_timeout              =*/ 0,    // second  0-> no timeout
/*.hdl                            =*/{0x00, 0x0063, 0x00, 0x00, 0x00},
                                                        //[HANDLE_NUM_MAX];
/*.serv                           =*/ {0x00, UUID_SERVICE_BATTERY, 0x00, 0x00, 0x00},
/*.cha                            =*/ {0x00, UUID_CHARACTERISTIC_BATTERY_LEVEL, 0x00,
                                     0x00, 0x00},
/*.findme_locator_enable          =*/ 0,    // if 1 Find me locator is enable
/*.findme_alert_level             =*/ 0,    // alert level of find me
/*.client_grouptype_enable        =*/ 0,    // if 1 grouptype read can be used
/*.linkloss_button_enable         =*/ 0,    // if 1 linkloss button is enable
/*.pathloss_check_interval        =*/ 0,    // second
/*.alert_interval                 =*/ 0,    // interval of alert
/*.high_alert_num                 =*/ 0,    // number of alert for each interval
/*.mild_alert_num                 =*/ 0,    // number of alert for each interval
/*.status_led_enable              =*/ 1,    // if 1 status LED is enable
/*.status_led_interval            =*/ 1,    // second
/*.status_led_con_blink           =*/ 2,    // blink num of connection
/*.status_led_dir_adv_blink       =*/ 0,    // blink num of dir adv
/*.status_led_un_adv_blink        =*/ 2,    // blink num of undir adv
```

Here we can define the time ON/OFF of a LED that we can define later on.

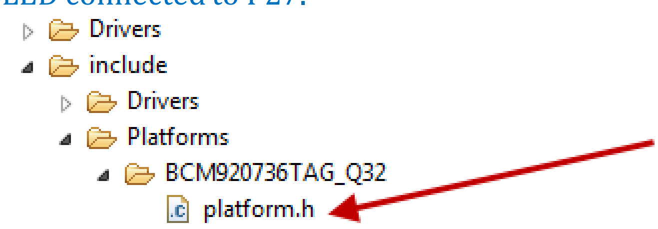
```
/*.led_on_ms                        =*/ 400,    // led blink on duration in ms
/*.led_off_ms                       =*/ 400,    // led blink off duration in ms
```

Here we can define a buzzer duration if the system has one. The EMRF-20737S-BOB does not have a Buzzer so we can just leave this as 0 or you could set it to a value and view the PWM output on P28/pin39.

```
/*.buz_on_ms                        =*/ 0,  // buzzer on duration in ms
/*.button_power_timeout             =*/ 0,    // seconds
/*.button_client_timeout            =*/ 0,    // seconds
/*.button_discover_timeout          =*/ 0,    // seconds
/*.button_filter_timeout            =*/ 0,    // seconds
#ifdef BLE_UART_LOOPBACK_TRACE
/*.button_uart_timeout              =*/ 15,   // seconds
#endif
};
```

3) PUART and GPIO Configuration

This next section we can define whether we want to enable the Peripheral UART(PUART) and/or various GPIO settings that are defined in the Broadcom BTLE Stack. The default values for these pins are defined in ***platform.h*** which is in the ***include/Platforms/BCM920736TAG_Q32*** folder as shown below. The EMRF-20737S-BOB makes use of the same Pushbutton and LED GPIO pins as are defined in the ***platform.h***. There is an additional Pushbutton connected to P4 and an additional LED connected to P27.



Here we can either enable or disable the PUART via the BTLE stack. My recommendation is to Configure this manually if you want to use it as you have more control over how it is configured. The PUART can also be used to output Debug messages. Using the PUART and outputting Debug Messages will be discussed in a follow on document and a code example provided.

```
// Following structure defines UART configuration
const BLE_PROFILE_PUART_CFG hello_sensor_puart_cfg =
{
   /*.baudrate     =*/ 115200,
   /*.txpin =*/ PUARTDISABLE | GPIO_PIN_UART_TX,
   /*.rxpin =*/ PUARTDISABLE | GPIO_PIN_UART_RX,
};
```

Here we can define the WP pin for the EEPROM internal to the Module. The WP pin definition should NOT be modified in the ***platform.h*** file as it is internally connected to P1. There is no way to change this unless you are doing a discrete SOC design. We also can define the default Pushbutton that will trigger an interrupt. This is defined as P0 in ***platform.h***. Also defined here is the default LED which is defined as P14. You could modify the Button and LED definitions in ***platform.h*** to make use of P4 for the button and P27 for the LED on the ***EMRF-20737S-BOB***. The Flag definitions further down are simply flags that get checked in the inner workings of the BTLE stack.

NOTE: I have commented out the Battery and Buzzer definitions. You could connect a battery to the system and feed the battery voltage into P15 if you so desire.

```
// Following structure defines GPIO configuration used by the application
const BLE_PROFILE_GPIO_CFG hello_sensor_gpio_cfg =
{
   /*.gpio_pin =*/
   {
     GPIO_PIN_WP,      // This need to be used to enable/disable NVRAM write protect
     GPIO_PIN_BUTTON, // Button GPIO is configured to trigger either direction of interrupt
     GPIO_PIN_LED,     // LED GPIO, optional to provide visual effects
     -1, //GPIO_PIN_BATTERY, // Battery monitoring GPIO. When it is lower than particular
                          // level, it will give notification to the application
     -1, //GPIO_PIN_BUZZER, // Buzzer GPIO, optional to provide audio effects
     -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 // other GPIOs are not used
   },
   /*.gpio_flag =*/
   {
     GPIO_SETTINGS_WP, GPIO_SETTINGS_BUTTON, GPIO_SETTINGS_LED,
     0, //GPIO_SETTINGS_BATTERY,
     0, //GPIO_SETTINGS_BUZZER,
     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
   }
};
```

4) Application Init()

Just as it sounds we will initialize the system. Note that here we are passing in a pointer to the Gatt_Database we defined earlier.

i. We define the size of the ***GATT_Database***
ii. We pass in a pointer to the ***BLE_PROFILE_CONFIG*** defined earlier
iii. We pass in a pointer to the Puart & GPIO config structures we just defined
iv. Last thing we do is we call ***hello_sensor_create()*** function.

```
// Application initialization
APPLICATION_INIT()
{
    bleapp_set_cfg((UINT8 *)hello_sensor_gatt_database,
          sizeof(hello_sensor_gatt_database),
          (void *)&hello_sensor_cfg,
          (void *)&hello_sensor_puart_cfg,
          (void *)&hello_sensor_gpio_cfg,
          hello_sensor_create);
}
```

5) hello_sensor_create()

This is where were we continue the system initialization and setup our callback functions we want to make use of.

```
void hello_sensor_create(void)
{
```

Here we instantiate a BLE Profile Database PDU(Protocol Data Unit) for use later.

```
BLEPROFILE_DB_PDU db_pdu;
extern UINT32 blecm_configFlag ;
blecm_configFlag |= BLECM_DBGUART_LOG;
ble_trace0("hello_sensor_create()");
ble_trace0(bleprofile_p_cfg->ver);
```

Here we output the entire GATT Database. This is what you see when you RESET the device and see all the values initially being printed out to the terminal window. In SDK2.0.1 there are better descriptions of what these values are related to compared to SDK1.1 which supports the BCM20732/BCM20732S.

```
// dump the database to debug uart.
legattdb_dumpDb();
```

We now initialize the ***BLE_PROFILE*** structure that we define earlier and below that we initialize the GPIO settings we defined earlier. These functions are defined in ***bleprofile.h***.

```
bleprofile_Init(bleprofile_p_cfg);
bleprofile_GPIOInit(bleprofile_gpio_p_cfg);
```

All this function does is it calls the ***blebat_init()*** function which sets up the reading the battery voltage with an A/D. Since I don't have a battery attached to the system now I simply comment out the ***blebat_init()*** function call inside of the ***hello_sensor_database_init()*** function.

```
hello_sensor_database_init();   //load handle number
```

Here is where we register/setup the Callback functions for various events such as when a Connection is made, a Disconnection Even happens, or the Advertisements timeout. We will inspect these functions later. ***bleprofile_regAppEvtHandler()*** is defined in ***blecm.h***.

```
// register connection up and connection down handler.
bleprofile_regAppEvtHandler(BLECM_APP_EVT_LINK_UP, hello_sensor_connection_up);
bleprofile_regAppEvtHandler(BLECM_APP_EVT_LINK_DOWN, hello_sensor_connection_down);
bleprofile_regAppEvtHandler(BLECM_APP_EVT_ADV_TIMEOUT, hello_sensor_advertisement_stopped);
```

The next 2 function calls are for when the encryption status has changed and the Secure Pairing/Bonding has occurred.

```
// handler for Encryption changed.
blecm_regEncryptionChangedHandler(hello_sensor_encryption_changed);
// handler for Bond result
lesmp_regSMPResultCb((LESMP_SINGLE_PARAM_CB) hello_sensor_smp_bond_result);
```

This is a callback function that is called when the Client writes to the slave/server. In this example this would occur when the client/master configures a Notification/Indication or writes a value into the ***HELLO_CHARACTERISTIC_CONFIG*** value to blink the LED a specified amount.

```
// register to process client writes
legattdb_regWriteHandleCb((LEGATTDB_WRITE_CB)hello_sensor_write_handler);
```

This is the callback for the Pushbutton Interrupt.

```
// register interrupt handler
bleprofile_regIntCb((BLEPROFILE_SINGLE_PARAM_CB) hello_sensor_interrupt_handler);
```

This is not in the 'out of box' hello_sensor application but it is included in my example as it is needed to make use of putting the BCM20737S into Deep Sleep. The ***devlpm_init()*** initializes the low power mode and the ***devlpm_enableWakeFrom()*** specifies to wake from Deep Sleep from a GPIO interrupt. You can also wake up from a 'timed wake' from either the internal 128kHz LPO or an external 32kHz XTAL. Timed wakeups will be covered in a future Appnote with a code example. ***devlpm_init()*** and ***devlpm_enableWakeFrom()*** are defined in ***devicelpm.c***.

```
// If power save timeout is not enabled, enable device LPM
// If powersave_timeout is enabled, the FW would have enabled it already along with a
// number of other things. This is needed for the app to be able to register a
// callback that is invoked to participate in sleep decisions.
if(!hello_sensor_cfg.powersave_timeout)
{
      ble_trace0("Call devlpm_init and Config GPIO Wakeup\n");
      devlpm_init();
      devlpm_enableWakeFrom(DEV_LPM_WAKE_SOURCE_GPIO);
}
```

Here we are registering the Callbacks for the fine timer(fast timer) and the 1sec default timer. We then start the timers. After the ***ble_profileStartTimer()*** is called you will start seeing the Timer printouts on the terminal window.

```
bleprofile_regTimerCb(hello_sensor_fine_timeout, hello_sensor_timeout);
bleprofile_StartTimer();
```

The next section starts the advertisements. This will be covered in more detail in a future Appnote using the Beacon example that is in SDK2.0.1 and beyond which supports the BCM20736S and BCM20737S.

```
// Read value of the service from GATT DB.
bleprofile_ReadHandle(HANDLE_HELLO_SENSOR_SERVICE_UUID, &db_pdu);
ble_tracen((char *)db_pdu.pdu, db_pdu.len);
if (db_pdu.len != 16)
{
      ble_trace1("hello_sensor bad service UUID len: %d\n", db_pdu.len);
}
else
{
      BLE_ADV_FIELD adv[3];
       // flags
       adv[0].len     = 1 + 1;
       adv[0].val     = ADV_FLAGS;
```

```
        adv[0].data[0] = LE_LIMITED_DISCOVERABLE | BR_EDR_NOT_SUPPORTED;

        adv[1].len     = 16 + 1;
        adv[1].val     = ADV_SERVICE_UUID128_COMP;
        memcpy(adv[1].data, db_pdu.pdu, 16);

        // name
        adv[2].len      = strlen(bleprofile_p_cfg->local_name) + 1;
        adv[2].val      = ADV_LOCAL_NAME_COMP;
        memcpy(adv[2].data, bleprofile_p_cfg->local_name, adv[2].len - 1);

        bleprofile_GenerateADVData(adv, 3);
    }
    blecm_setTxPowerInADV(0);
    bleprofile_Discoverable(HIGH_UNDIRECTED_DISCOVERABLE, hello_sensor_remote_addr);
    ble_trace0("end AppInit \n");
} // end hello_sensor_create()
```

6) hello_sensor_connection_up()

This is the callback function for when a connection gets established.

```
void hello_sensor_connection_up(void)
{
    UINT8 writtenbyte;
    UINT8 *bda;
```

This function gets the handle value of the Client we have connected to.

```
    hello_sensor_connection_handle = (UINT16)emconinfo_getConnHandle();
```

As the inline comment indicates we are retrieving the BD Address of the Client and make a copy.

```
    // save address of the connected device and print it out.
    memcpy(hello_sensor_remote_addr, (UINT8 *)emconninfo_getPeerAddr(),
    sizeof(hello_sensor_remote_addr));
```

Here we are just printing out the BD Address of the Client to the terminal window.

```
    ble_trace3("hello_sensor_connection_up: %08x%04x %d\n",
        (hello_sensor_remote_addr[5] << 24) + (hello_sensor_remote_addr[4] << 16) +
        (hello_sensor_remote_addr[3] << 8) + hello_sensor_remote_addr[2],
        (hello_sensor_remote_addr[1] << 8) + hello_sensor_remote_addr[0],
        hello_sensor_connection_handle);
```

Since we have established a connection we can stop advertising so we stop adverstising.

```
    // Stop advertising
    bleprofile_Discoverable(NO_DISCOVERABLE, NULL);
```

We can also now stop the Connection Idle Timer. Since we have it disabled in the ***BLE_PROFILE_CFG*** this really has no effect but in a real application you would want to have a Connection Idle timer enabled.

```
    bleprofile_StopConnIdleTimer();
```

As the inline comment indicates since we have indicated we have indicated in the ***BLE_PROFILE_CFG*** that we want Encryption we start the encryption process.

```
// as we require security for every connection, we will not send any indications
// until encryption is done.
if (bleprofile_p_cfg->encr_required != 0)
{
   lesmp_sendSecurityRequest();
   return;
}
```

The next 2 lines of code we pull the Client BD Address and then store it in the hello_sensor_hostinfo structure which is defined just before the function prototypes at the top of the file.

```
// saving bd_addr in nvram
bda =(UINT8 *)emconninfo_getPeerAddr();

memcpy(hello_sensor_hostinfo.bdaddr, bda, sizeof(BD_ADDR));
```

The next 2 lines of code simply initialize the elements of a structure that stores the ***characteristic_client_configuration*** which is whether or not Indications or Notifications are enabled and then finally the number of Blinks that are stored from either the Pushbutton press or from the GUI. This structure is defined before the function prototypes.

```
hello_sensor_hostinfo.characteristic_client_configuration = 0;
hello_sensor_hostinfo.number_of_blinks = 0;
```

Now we take the BD Address of the Client/Master that was stored earlier in the structure and store it in the internal EEPROM.

```
writtenbyte = bleprofile_WriteNVRAM(VS_BLE_HOST_LIST, sizeof(hello_sensor_hostinfo),
   (UINT8 *)&hello_sensor_hostinfo);

ble_trace1("NVRAM write:%04x\n", writtenbyte);
```

Here we call the encryption_changed function since the secure connection has started.

```
    hello_sensor_encryption_changed(NULL);
} // end hello_sensor_connection_up()
```

7) hello_sensor_connection_down()

This is the callback function for when the connection is closed or lost. The user is to decide whether to attempt to start advertising again to reconnect or to do something else such as possibly go into a Low Power Mode. I have added some code to go into Deep Sleep if the variable ***hello_sensor_stay_connected*** is not set.

```
void hello_sensor_connection_down(void)
{
```

Here we are just printing out to the terminal window the address and handle of the client that we lost the connection to.

```
    ble_trace3("hello_sensor_connection_down:%08x%04x handle:%d\n",
          (hello_sensor_remote_addr[5] << 24) + (hello_sensor_remote_addr[4] << 16) +
          (hello_sensor_remote_addr[3] << 8) + hello_sensor_remote_addr[2],
          (hello_sensor_remote_addr[1] << 8) + hello_sensor_remote_addr[0],
          hello_sensor_connection_handle);
```

Here we are clearing the BD Address of the client and clearing the handle value since the connection was lost.

```
    memset (hello_sensor_remote_addr, 0, 6);
    hello_sensor_connection_handle = 0;
```

This section will either restart the Advertising to try and re-connect OR if the ***hello_sensor_stay_connected*** variable is set to 0 we go into Deep Sleep at just over 1uA at 1.8V and a bit higher at higher voltages. The device can be woken up by any GPIO that has been configured as an interrupt such as the P0 button. Keep in mind P0 is configured as an interrupt in ***platform.h***.

```
// If we are configured to stay connected, disconnection was caused by the
// peer, start low advertisements, so that peer can connect when it wakes up.
if (hello_sensor_stay_connected)
{
    bleprofile_Discoverable(LOW_UNDIRECTED_DISCOVERABLE,
    hello_sensor_hostinfo.bdaddr);

    ble_trace2("ADV start: %08x%04x\n", (hello_sensor_hostinfo.bdaddr[5] << 24 ) +
                                       (hello_sensor_hostinfo.bdaddr[4] <<16) +
                (hello_sensor_hostinfo.bdaddr[3] << 8 ) + hello_sensor_hostinfo.bdaddr[2],
                (hello_sensor_hostinfo.bdaddr[1] << 8 ) + hello_sensor_hostinfo.bdaddr[0]);
 }
 else
 {
     ble_trace0("Entering DeepSleep - Connection Lost \n");
     bleapputils_delayUs(500);
     bleprofile_PrepareHidOff();
 }
} // end hello_sensor_connnection_down()
```

8) hello_sensor_advertisement_stopped()

This is the callback function that gets called if there has not been a connection established after both the High and the Low Advertising Intervals have expired. Keep in mind it is ONLY after BOTH the High and Low Advertising Intervals have expired that this callback will be called unless of course you only have one of them specified. I have again added the capability to go into Deep Sleep if the ***hell_sensor_stay_connected*** variable is set to 0. Otherwise we will start the Advertisements again.

```
void hello_sensor_advertisement_stopped(void)
{
   ble_trace0("ADV stop!!!!\n");
   // If we are configured to stay connected, disconnection was caused by the
   // peer, start low advertisements, so that peer can connect when it wakes up.
   if (hello_sensor_stay_connected)
   {
     bleprofile_Discoverable(LOW_UNDIRECTED_DISCOVERABLE, hello_sensor_hostinfo.bdaddr);

     ble_trace2("ADV start: %08x%04x\n",(hello_sensor_hostinfo.bdaddr[5] << 24 ) +
                                        (hello_sensor_hostinfo.bdaddr[4] <<16) +
                (hello_sensor_hostinfo.bdaddr[3] << 8 ) + hello_sensor_hostinfo.bdaddr[2],
                (hello_sensor_hostinfo.bdaddr[1] << 8 ) + hello_sensor_hostinfo.bdaddr[0]);
   }
   else
   {
     ble_trace0("Entering DeepSleep - AdvStopped \n"); bleapputils_delayUs(500);
     bleprofile_Discoverable(NO_DISCOVERABLE, NULL);
     bleprofile_PrepareHidOff();      //Puts device into DeepSleep ~1.33uA
   }
}
```

9) hello_sensor_timeout()/fine_timeout()

These are the callback functions for the fine timer and the 1 second tick timer. Keep in mind the fine_timer timeout value is set in the ***BLE_PROFILE_CFG*** structure. The fine timer can be configured for 12ms to 1sec timeout. Anything greater than 1000 will result in effectively in a 1 second timeout.

```
void hello_sensor_timeout(UINT32 arg)
{
   ble_trace1("hello_sensor_timeout:%d\n", hello_sensor_timer_count);

   switch(arg)
   {
      case BLEPROFILE_GENERIC_APP_TIMER:
      {
            hello_sensor_timer_count++;
      }
   break;
   }
}
```

NOTE: We are using the fine_timer to check the Pushbutton defined in the ***BLE_PROFILE_CFG*** and ***platform.h***.

```
void hello_sensor_fine_timeout(UINT32 arg)
{
   hello_sensor_fine_timer_count++;
```

If you want to see the output of this timer you can put this line of code in your hello_sensor example.

```
   // ble_trace1("hello_sensor_fine_timeout:%d", hello_sensor_fine_timer_count);

   // button control
   bleprofile_ReadButton();
}
```

10) hello_sensor_smp_bond_result()

This callback function is setup in the ***hello_sensor_create()*** function and is called during the bonding process. If the bonding/pairing is successful we store the Client/Master's BD Address in EEPROM and we initialize the ***hello_sensor_hostinfo*** structure mentioned earlier.

```
void hello_sensor_smp_bond_result(LESMP_PARING_RESULT     result)
{
    ble_trace1("hello_sample, bond result %02x\n", result);
```

On the ***EMRF-20737S-BOB*** you could comment this line out as we don't have a buzzer or if you want to look at the signal you can ensure the Buzzer is defined in platform.h and ensure it gets defined in the ***BLE_PROFILE_GPIO_CFG*** structure. The default Port if P28/pin39 as defined in ***platform.h.*** You could choose a different PWM pin also if you like. If you have a scope you will see a PWM waveform on P28/pin39.

```
    // do some noise
    bleprofile_BUZBeep(bleprofile_p_cfg->buz_on_ms);
```

LESMP_PAIRING_RESULT_BONDED is a typedef enum and is defined in ***lesmp.h***.

```
    if (result == LESMP_PAIRING_RESULT_BONDED)
    {
            // saving bd_addr in nvram
            UINT8 *bda;
            UINT8 writtenbyte;

            bda =(UINT8 *)emconninfo_getPeerAddr();
            memcpy(hello_sensor_hostinfo.bdaddr, bda, sizeof(BD_ADDR));
            hello_sensor_hostinfo.characteristic_client_configuration = 0;
            hello_sensor_hostinfo.number_of_blinks = 0;

            writtenbyte = bleprofile_WriteNVRAM(VS_BLE_HOST_LIST,
            sizeof(hello_sensor_hostinfo), (UINT8 *)&hello_sensor_hostinfo);
            ble_trace1("NVRAM write:%04x\n", writtenbyte);
    }
}
```

11) hello_sensor_encryption_changed()

This function is called from the ***hello_sensor_connection_up()*** function to indicate to the BTLE stack that encryption has be set and if the client/master has registered for notifications/indications we can send them out now.

The ***HCI_EVT_HDR*** struct is defined in ***cfa.h.*** The handler function is setup in ***hello_sensor_create()***.

```
void hello_sensor_encryption_changed(HCI_EVT_HDR *evt)
{
```

As a reminder ***BLEPROFILE_DB_PDU*** is defined in ***bleprofile.h***

```
BLEPROFILE_DB_PDU db_pdu;

ble_trace0("hello_sample, encryption changed\n");
```

For the EMRF-20737S there is not a Buzzer so you could either comment this out or leave it in place and view the PWM output on the Pin that is assigned in ***platform.h*** for the Buzzer which is P28/pin 39.

```
bleprofile_BUZBeep(bleprofile_p_cfg->buz_on_ms);
```

Here we are pulling the stored client/master BD Address from NVRAM. ***VS_BLE_HOST_LIST*** is defined in ***stacknvram.h***. The function ***bleprofile_ReadNVRAM*** is defined in ***bleprofile.h***. The ReadNVRAM function has the following input parameters.

UINT8 ID number of NVRAM(0-0x6F)	= VS_BLE_HOST_LIST(0x70)
UINT8 itemLength	= sizeof(hello_sensor_hostinfo)
UINT8* payload	= &hello_sensor_hostinfo

```
// Connection has been encrypted meaning that we have correct/paired device
// restore values in the database
bleprofile_ReadNVRAM(VS_BLE_HOST_LIST, sizeof(hello_sensor_hostinfo),
    (UINT8*)&hello_sensor_hostinfo);
```

Here we are loading the client configuration into a PDU packet and then later writing it into the descriptor value. This value determines whether or not Notifications or Indications have been enabled by the Client/Master.

```
// Need to setup value of Client Configuration descriptor in our database because
// peer might decide to read and stack sends answer without asking application.
db_pdu.len = 2;
db_pdu.pdu[0] = hello_sensor_hostinfo.characteristic_client_configuration & 0xff;
db_pdu.pdu[1] = (hello_sensor_hostinfo.characteristic_client_configuration >> 8) & 0xff;
```

Here we are updating the GATT database for the handle specified as ***HANDLE_HELLO_SENSOR_CLIENT_CONFIGURATION_DESCRIPTOR***. The ***bleprofile_WriteHandle()*** function is defined in ***bleprofile.h***. This function provides a means for the application to write to the GATT database.

```
bleprofile_WriteHandle(HANDLE_HELLO_SENSOR_CLIENT_CONFIGURATION_DESCRIPTOR,
                             &db_pdu);
```

Similar to what we have done above we are going to store the number of 'blinks' which will be 0 upon the initial connection. On the terminal window you will see this output from the ***ble_trace4*** debug output below…

```
00:24:52 - blecm evt handler:
00:24:52 - 080400400001
00:24:52 - Link Encrypted
00:24:52 - hello_sample, encryption changed 000272c6a2c1
00:24:52 -
00:24:52 -
00:24:52 -  permission check retCode = 00
00:24:52 -
00:24:52 -  permission check retCode = 00
00:24:52 - EncOn 000272c6a2c1 client_configuration:0001 blinks:5
00:24:52 -
```

```
// Setup value of our configuration in GATT database db_pdu.len = 1;
db_pdu.pdu[0] = hello_sensor_hostinfo.number_of_blinks;
bleprofile_WriteHandle(HANDLE_HELLO_SENSOR_CONFIGURATION, &db_pdu);

ble_trace4("EncOn %08x%04x client_configuration:%04x blinks:%d\n",
    (hello_sensor_hostinfo.bdaddr[5] << 24) + (hello_sensor_hostinfo.bdaddr[4] << 16) +
     (hello_sensor_hostinfo.bdaddr[3] << 8) + hello_sensor_hostinfo.bdaddr[2],
     (hello_sensor_hostinfo.bdaddr[1] << 8) + hello_sensor_hostinfo.bdaddr[0],
     hello_sensor_hostinfo.characteristic_client_configuration,
     hello_sensor_hostinfo.number_of_blinks);
```

Just as the comments indicate we will send out any outstanding messages that may be present before the encrypted connection has been established. Keep in mind that Indications require an ACK from the client/master.

```
// If there are outstanding messages that we could not send out because
// connection was not up and/or encrypted, send them now.   If we are sending
// indications, we can send only one and need to wait for ack.
while ((hello_sensor_num_to_write != 0) && !hello_sensor_indication_sent)
{
    hello_sensor_num_to_write--;
    hello_sensor_send_message();
}
```

If the ***hello_sensor_stay_connected*** variable is set to 0 then we will start a connection idle timer to disconnect after the connection has been idle for the time specified in the ***BLE_PROFILE_CFG*** structure. As recommended for debug purposes we typically would set the ***.con_idle_timeout*** to 0 in the ***BLE_PROFILE_CFG*** so that we don't lose the connection. You could set this to a specified time in Seconds if you wanted to disconnect after sending data and the connection has been idle for the specified time.

```
// If configured to disconnect after delivering data, start idle timeout to do
// disconnection
if (!hello_sensor_stay_connected && !hello_sensor_indication_sent)
{
   bleprofile_StartConnIdleTimer(bleprofile_p_cfg->con_idle_timeout,
                                 bleprofile_appTimerCb);
   return;
}
```

As indicated in the comments we are sending a connection update to the client/master to slow down the polling rate to save power.

```
// We are done with initial settings, and need to stay connected.  It is a good
// time to slow down the pace of master polls to save power. Following request asks
// host to setup polling every 100-500 msec, with link supervision timeout 7
// seconds.

    bleprofile_SendConnParamUpdateReq(80, 400, 0, 700);
}  // end hello_sensor_encryption_changed()
```

12) **hello_sensor_send_message()**

This function is called from h***ello_sensor_encryption_changed()***, ***hello_sensor_indication_cfm()***, and ***hello_sensor_interrupt_handler()***. This function sends out a message if the client has indicated it wants to receive Indications or Notifications. In the case of the hello_sensor app this would entail a button press but in a real world application this would involve some sort of data that needs to be updated to the client/master if it has changed.

```
// Check if client has registered for notification and indication and send message if
// appropriate
void hello_sensor_send_message(void)
{
    BLEPROFILE_DB_PDU db_pdu;
```

If the value for the ***characteristic_client_configuration*** is 0 it means that the client/master has not registered/told the peripheral/slave that it wants to receive Notifications(= 1) or Indications(= 2). So we will simply return even if this function is called.

```
    // If client has not registered for indication or notification, do not need to do
    // anything
    if (hello_sensor_hostinfo.characteristic_client_configuration == 0)
      return;
```

Here we are reading the value that has been stored into the HELLO_CHARACTERISTIC_NOTIFY and is what you would see being incremented from the Pushbutton which is the 7th byte of 'Hello X' where X is the 7th byte that gets incremented if you push the pushbutton.

```
    // Read value of the characteristic to send from the GATT DB.
    bleprofile_ReadHandle(HANDLE_HELLO_SENSOR_VALUE_NOTIFY, &db_pdu);
    ble_tracen((char *)db_pdu.pdu, db_pdu.len);
```

Here we Logical AND the ***characteristic_client_configuration*** to determine if it is a Notification or if it is not it should be an Indication which is handled in the 'else' part of the statement. Note that the ***CCC_NOTIFICATION*** is defined in ***bleprofile.h*** and is defined as 0x01 whereas an Indication is defined as ***CCC_INDICATION*** and is defined as 0x02. The ***bleprofile_sendNotification()*** and ***bleprofile_sendIndication()*** functions are defined also in ***bleprofile.h***.

```
    if (hello_sensor_hostinfo.characteristic_client_configuration & CCC_NOTIFICATION)
    {
      bleprofile_sendNotification(HANDLE_HELLO_SENSOR_VALUE_NOTIFY, (UINT8*)db_pdu.pdu,
                                  db_pdu.len);
      ble_trace0("CCC_Notification Sent\n");
    }
```

```
else
{
  if (!hello_sensor_indication_sent)
  {
         hello_sensor_indication_sent = TRUE;
         ble_trace0("Hello_sens_indication_cfm sent\n");
```

NOTE: If it is an Indication there is a Callback function for the ACK/Confirmation from the client/master. This function is detailed later in this document. ***bleprofile_sendIndication()*** is defined in ***bleprofile.h.***

```
         bleprofile_sendIndication(HANDLE_HELLO_SENSOR_VALUE_NOTIFY,
                        (UINT8*)db_pdu.pdu, db_pdu.len, hello_sensor_indication_cfm);
   }
  }
} // end hello_sensor_send_message()
```

13) hello_sensor_write_handler()

This function processes all write commands from the client/master to the peripheral/slave.

```
// Process write request or command from peer device
```

NOTE: ***LEGATTDB_ENTRY_HDR*** is defined in ***legattdb.h***.

```
int hello_sensor_write_handler(LEGATTDB_ENTRY_HDR *p)
{
    UINT8   writtenbyte;
```

The legattdb_xxxx functions are defined in ***legattdb.h***

```
    UINT16 handle  = legattdb_getHandle(p);
    int     len    = legattdb_getAttrValueLen(p);
    UINT8   *attrPtr = legattdb_getAttrValue(p);
```

Again the EMRF-20737S does not have a Buzzer on it so you can either comment this line out or leave it in place and if you would like view the PWM output that is intended to drive the buzzer on P28/pin 39.

```
    // do some noise
    bleprofile_BUZBeep(bleprofile_p_cfg->buz_on_ms);
```

Here we are reading the client/master BD Address to ensure it is the device we are actually paired to that is attempting to write to the peripheral/slave.

```
    // make sure that it is the paired device which is trying to write
    // read BDADDR of the "paired device" from the NVRAM and compare with connected
    bleprofile_ReadNVRAM(VS_BLE_HOST_LIST, sizeof(hello_sensor_hostinfo),
                            (UINT8*)&hello_sensor_hostinfo);
```

This is where we compare what is stored in the ***VS_BLE_HOST_LIST*** to what we have stored in the ***hello_sensor_remote_addr***. Since the BD Address is 6bytes(48bits) we compare all 6bytes. If the compare doesn't equal 0 we return and put out a Debug message indicating that it is the wrong host handle. If not we print out the handle of the client/master.

```
if (memcmp(hello_sensor_remote_addr, hello_sensor_hostinfo.bdaddr, 6) != 0)
{
  ble_trace1("hello_sensor_write_handler: wrong host handle %04x\n", handle);
        return 0;
}
ble_trace1("hello_sensor_write_handler: handle %04x\n", handle);
```

Here we determine if what is being wrote to the peripheral/slave is the ***HELLO_SENSOR_CLIENT_CONFIGURATION_DESCRIPTOR*** which is 2bytes. If so the client/master is writing to indicate it wants to receive Notifications(0x01) or Indications(0x02). ***len*** is defined above by `int len = legattdb_getAttrValueLen(p);`

```
// By writing into Characteristic Client Configuration descriptor
// peer can enable or disable notification or indication
if ((len == 2) && (handle ==
        HANDLE_HELLO_SENSOR_CLIENT_CONFIGURATION_DESCRIPTOR))
{
  hello_sensor_hostinfo.characteristic_client_configuration = attrPtr[0] +
              (attrPtr[1] << 8);
  ble_trace1("hello_sensor_write_handler: client_configuration %04x\n",
  hello_sensor_hostinfo.characteristic_client_configuration);
}
```

If what is being wrote to the peripheral/slave from the client/master is not whether it wants to accept Notification/Indications it will be to update the number of blinks which is the ***HELLO_SENSOR_CONFIGURATION***. **NOTE:** This and the ***HELLO_SENSOR_CLIENT_CONFIGURATION_DESCRIPTOR*** are both defined in the GATT database at the beginning of the app.

```
// User can change number of blinks to send when button is pushed
else if ((len == 1) && (handle == HANDLE_HELLO_SENSOR_CONFIGURATION))
{
  hello_sensor_hostinfo.number_of_blinks = attrPtr[0];
  if (hello_sensor_hostinfo.number_of_blinks != 0)
  {
        bleprofile_LEDBlink(250, 250, hello_sensor_hostinfo.number_of_blinks);
        ble_trace0("LED Blink write_handler\n");
  }
}
```

This is for the case that the write has the incorrect length and handles the fallout by printing a Debug message indicating so.

```
else
{
   ble_trace2("hello_sensor_write_handler: bad write len:%d handle:0x%x\n", len,
                        handle);
    return 0x80;
}
```

Here we are store the what the client/master wrote to the peripheral/slave device in NVRAM/EEPROM that is internal on the BCM20737S.

```
// Save update to NVRAM.    Client does not need to set it on every connection.
writtenbyte = bleprofile_WriteNVRAM(VS_BLE_HOST_LIST, sizeof(hello_sensor_hostinfo),
          (UINT8 *)&hello_sensor_hostinfo);
ble_trace1("hello_sensor_write_handler: NVRAM write:%04x\n", writtenbyte);

   return 0;
} // end hello_sensor_write_handler()
```

14) hello_sensor_interrupt_handler()

This function handles the interrupts that are generated from the pushbutton. Keep in mind in this example the GPIO that is attached to the pushbutton is defined in ***platform.h***. You can also manually indicate GPIO to be interrupts. Note that this function is defined as a callback in the ***hello_sensor_create*** function.

```
// Three Interrupt inputs (Buttons) can be handled here.
// If the following value == 1, Button is pressed. Different than initial value.
// If the following value == 0, Button is depressed. Same as initial value.
// Button1 : value&0x01
// Button2 : (value&0x02)>>1
// Button3 : (value&0x04)>>2
void hello_sensor_interrupt_handler(UINT8 value)
{
      BLEPROFILE_DB_PDU db_pdu;
```

Determine which button that was configured was pressed.

```
      ble_trace3("(INT)But1:%d But2:%d But3:%d\n", value&0x01, (value& 0x02) >> 1,
                  (value & 0x04) >> 2);
```

Blink the stored 'number of blinks'.

```
      // Blink as configured
      bleprofile_LEDBlink(250, 250, hello_sensor_hostinfo.number_of_blinks);
      ble_trace0("LED Blink interrupt_handler\n");
```

Here we store the amount of ‘button presses’ in the last byte of Hello X. You can see that we are incrementing the 7th byte of the array ‘H’, ‘e’, ‘l’, ‘l’, ‘o’, ‘ ‘, ‘0’. If the last byte is greater than 9 we reset to 0. So in the hello_sensor GUI you see Hello X increment from 0-9 and then rollover to 0 again. After the last byte is incremented it is Stored back into the ***HELLO_SENSOR_VALUE_NOTIFY*** via the ***bleprofile_WriteHandle()*** function.

```
// keep number of the button pushes in the last byte of the Hello %d message. That
// will guarantee that if client reads it, it will have correct data.
bleprofile_ReadHandle(HANDLE_HELLO_SENSOR_VALUE_NOTIFY, &db_pdu);
ble_tracen((char *)db_pdu.pdu, db_pdu.len);
db_pdu.pdu[6]++;
if (db_pdu.pdu[6] > '9')
      db_pdu.pdu[6] = '0';
bleprofile_WriteHandle(HANDLE_HELLO_SENSOR_VALUE_NOTIFY, &db_pdu);
```

For every button press we increment the amount of messages that we need to send to the client/master.

```
// remember how many messages we need to send
hello_sensor_num_to_write++;
```

The ***hello_sensor_connection_handle*** is set in the ***hello_sensor_connection_up()*** function and is cleared in the ***hello_sensor_connection_down()*** function. So if it is 0 we have either lost the connection or it was never established so we start advertising to establish the connection. Note that ***hello_sensor_remote_addr*** is cleared if ***hello_sensor_connection_down()*** has been called after the connection has been closed.

```
// If connection is down, we need to start advertisements, so that client can connect
if (hello_sensor_connection_handle == 0)
{
      bleprofile_Discoverable(HIGH_UNDIRECTED_DISCOVERABLE, hello_sensor_remote_addr);

      ble_trace2("ADV start high: %08x%04x\n",(hello_sensor_hostinfo.bdaddr[5] << 24) +
              (hello_sensor_hostinfo.bdaddr[4] << 16) +
              (hello_sensor_hostinfo.bdaddr[3] << 8) + hello_sensor_hostinfo.bdaddr[2],
              (hello_sensor_hostinfo.bdaddr[1] << 8) + hello_sensor_hostinfo.bdaddr[0]);
      return;
}
```

If we have dropped to here the connection is up and the ***hello_sensor_connection_handle*** is therefore not 0. We can now send the Notifications or Indications to the client/master. Keep in mind indications require an ACK from the client/master. The ***hello_sensor_indication_sent*** variable is set in the ***hello_sensor_send_message()*** function and is cleared in the ***hello_sensor_indication_cfm()*** function.

```
// Connection is up. Send message if client is registered to receive indication
// or notification. After we sent an indication we need to wait for the ack before
// we can send anything else
while ((hello_sensor_num_to_write != 0) && !hello_sensor_indication_sent)
{
      hello_sensor_num_to_write--;
      hello_sensor_send_message();
}
```

If ***hello_sensor_stay_connected*** is set to 0 and we have sent all the messages we will start the Connection Idle Timer to disconnect after the specified time.

```
    // if we sent all messages, start connection idle timer to disconnect
    if (!hello_sensor_stay_connected && !hello_sensor_indication_sent)
    {
          bleprofile_StartConnIdleTimer(bleprofile_p_cfg->con_idle_timeout,
                 bleprofile_appTimerCb);
    }
} // end hello_sensor_interrupt_handler()
```

15) **hello_sensor_indication_cfm()**

This is the callback function indicated from the function ***bleprofile_sendIndication()*** which is in ***hello_sensor_send_message()***. This is the callback for the ACK from the client/master when an indication is sent out.

```
// process indication confirmation. If client wanted us to use indication instead of
// notifications we have to wait for confirmation after every message sent.   For
// example if user pushed button twice very fast we will send first message, wait for
// confirmation, send second message, wait for confirmation and if configured start
// idle timer only after that.
void hello_sensor_indication_cfm(void)
{
```

If ***hello_sensor_indication_sent*** is 0 it is not the correct ACK from the client/master as ***hello_sensor_indication_sent*** would have been set to 1(TRUE) in ***hello_sensor_send_message()***.

```
    if (!hello_sensor_indication_sent)
    {
      ble_trace0("Hello: Wrong Confirmation!!!");
      return;
    }
```

We now clear the ***hello_sensor_indication_sent*** variable as we have received the ACK back from the client/host indicating that it has received the indication.

```
    hello_sensor_indication_sent = 0;
```

If there are still more indications to send go ahead and send them.

```
    // We might need to send more indications
    if (hello_sensor_num_to_write)
    {
      hello_sensor_num_to_write--;
      hello_sensor_send_message();
    }
```

Similar to before if ***hello_sensor_stay_connected*** is set to 0 and we have sent all the indicatations we will start the Connection Idle timer and disconnect after the specified time that the connection has been idle.

```
    // if we sent all messages, start connection idle timer to disconnect
    if (!hello_sensor_stay_connected && !hello_sensor_indication_sent)
    {
      bleprofile_StartConnIdleTimer(bleprofile_p_cfg->con_idle_timeout,
                  bleprofile_appTimerCb);
    }
} // end hello_sensor_indication_cfm()
```

16) **bleprofile_StartConnIdleTimer()**

This function stops the Connection Idle timer. This function would be called if we have set the ***hello_sensor_stay_connected*** variable to 0.

```
// Start connection idle timer if it is not running
void bleprofile_StartConnIdleTimer(UINT8 timeout, BLEAPP_TIMER_CB cb)
{
    if(emconinfo_getAppTimerId() < 0)
    {
        emconinfo_setIdleConnTimeout(timeout);
        blecm_startConnIdleTimer(cb);
        ble_trace1("profile:idletimer(%d)", timeout);
    }
} //end bleprofile_StartConnIdleTimer()
```

17) **bleprofile_StopConnIdleTimer()**

This function stops the Connection Idle timer. This function would be called if we have set the ***hello_sensor_stay_connected*** variable to 0. This function is called in ***hello_sensor_connection_up()***.

```
// Stop connection idle timer if it is running
void bleprofile_StopConnIdleTimer(void)
{
    if(emconinfo_getAppTimerId() >= 0)
    {
        blecm_stopConnIdleTimer();
        emconinfo_setAppTimerId(-1);
        ble_trace0("profile:idletimer stopped");
    }
} //end bleprofile_StopConnIdleTimer()
```

18) **bleprofile_SendConnParamUpdateReq()**

This function sends updated Connection parameters to the client/master.

```
// Send request to client to update connection parameters
void bleprofile_SendConnParamUpdateReq(UINT16 minInterval, UINT16 maxInterval,
            UINT16 slaveLatency, UINT16 timeout)
{
    if (minInterval > maxInterval)
    return;
```

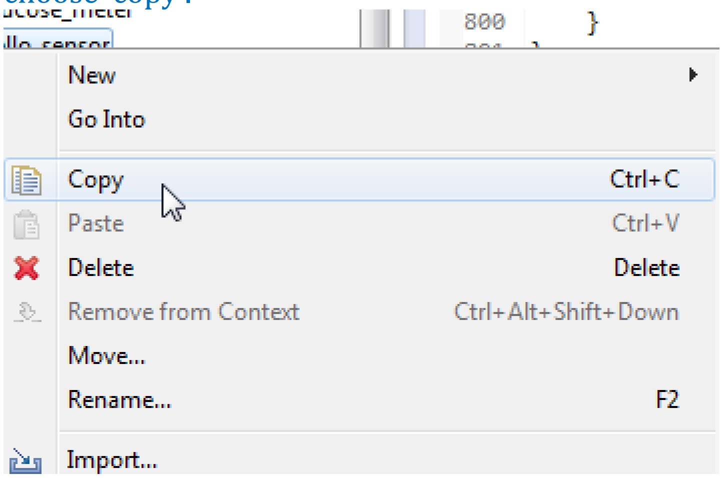
NOTE: ***lel2cap_sendConnParamUpdateReq()*** is defined in ***lel2cap.h***.

```
    lel2cap_sendConnParamUpdateReq(minInterval, maxInterval, slaveLatency, timeout);
}
```

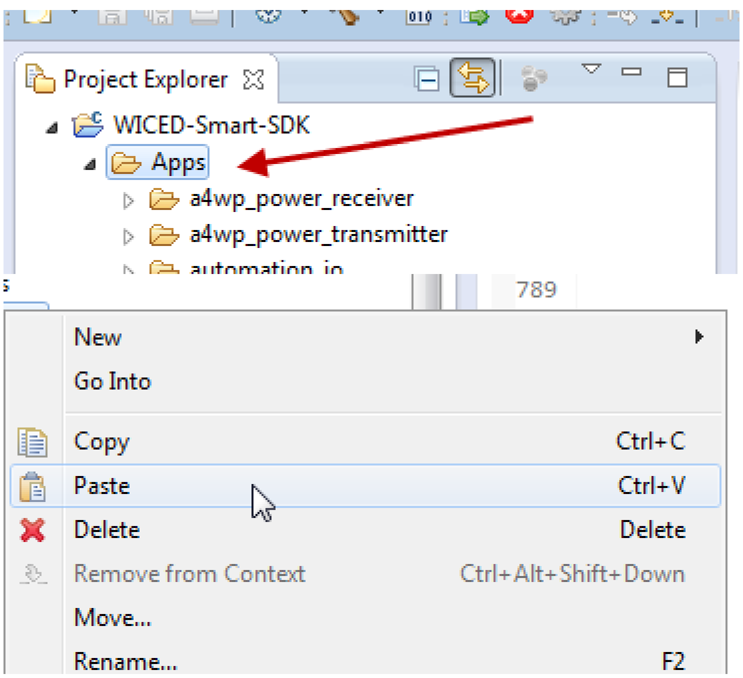
Section 5: Create my own Project

This section will give you a quick overview of how you can create your own project. The easiest way to start is to simply copy one of the existing projects and then you can modify the files and/or add your own files. This example will make use of the same ***hello_sensor*** App we have learned about and copy it to our own project directory so we can start on our own firmware development.

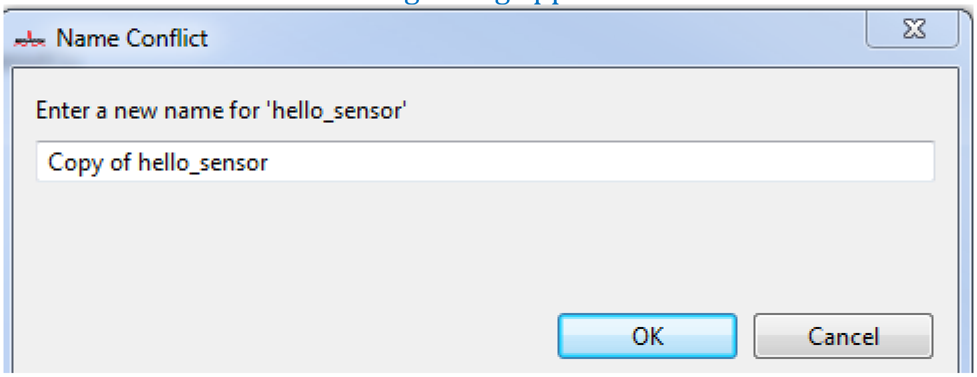
1) First we start out by simply copying the ***hello_sensor*** project directory and paste it into a new directory in Eclipse as shown below. To do this simply right-click on the ***hello_sensor*** folder and choose ‘copy’.



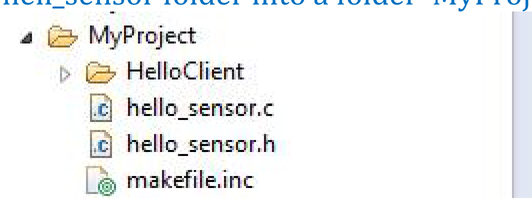
2) Now go to the top of the **APPS** folder and right-click and then choose ‘paste’



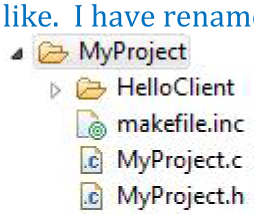
3) You will now see the following dialog appear…



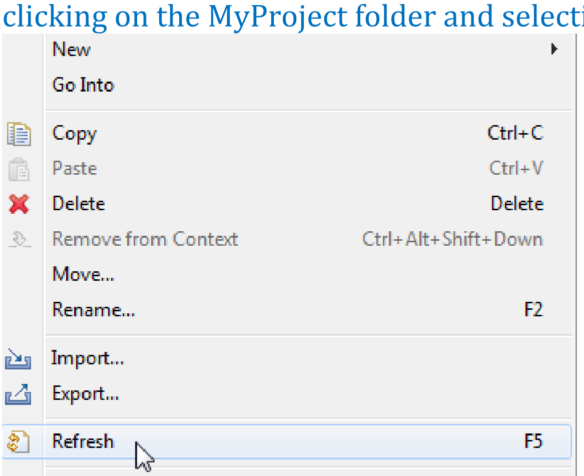
We just need to rename the project to our own name such as MyProject. It will place a copy of the hell_sensor folder into a folder 'MyProject' as shown below.



At this point I would recommend renaming the files so that you don't get confused which ones you are actually working on as you make changes. You could do this in Windows Explorer by going to the SDK2.x.x install directory and finding the folder in the 'Apps' directory or simply open the files and go to the main Menu bar and choosing File->Save As… and rename them to whatever you would like. I have renamed them MyProject.c and MyProject.h as shown below.



NOTE: If you modify the names in Windows Explorer and come back to Eclipse you may need to do a 'Refresh' of the directory to have the files show up. You can do this by either hitting F5 or by right-clicking on the MyProject folder and selecting 'Refresh'.



4) Now we need to modify the makefile.inc file so that it knows what files it needs to compile. If you open the makefile.inc now simply replace the hello_sensor.c file with MyProject.c or whatever you have renamed the files to as shown below…

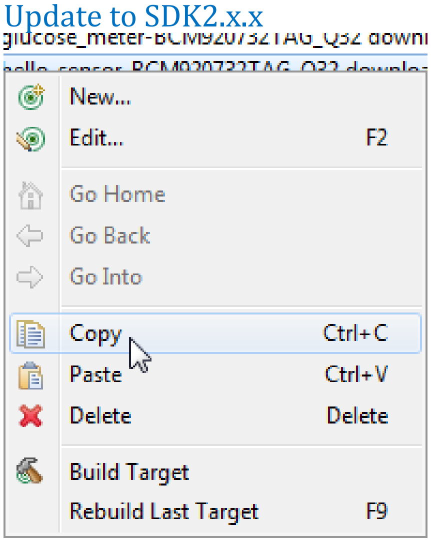
makefile.mk

```
#
# Copyright 2014, Broadcom Corporation
# All Rights Reserved.
#
# This is UNPUBLISHED PROPRIETARY SOURCE CODE of Broadcom Corporation;
# the contents of this file may not be disclosed to third parties, copied
# or duplicated in any form, in whole or in part, without the prior
# written permission of Broadcom Corporation.
#

########################################################################
# Add Application sources here.
########################################################################
APP_SRC = MyProject.c

########################################################################
################ DO NOT MODIFY FILE BELOW THIS LINE ####################
########################################################################
```

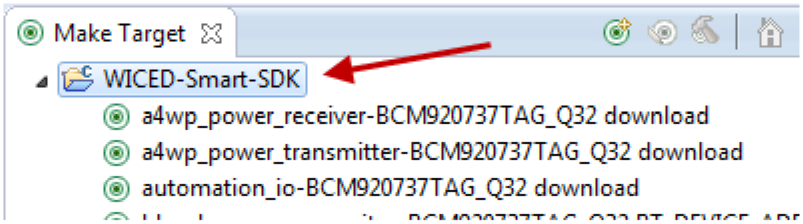
5) One other thing we need to do is modify in MyProject.c the header file name hello_sensor.h to MyProject.h as shown below… **NOTE:** Make sure to Manually SAVE the file!!

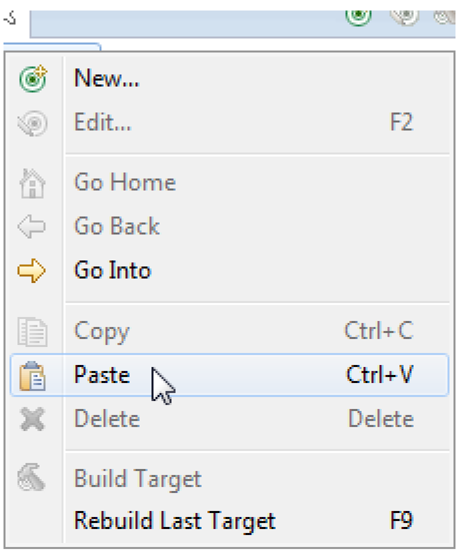
```
*/
#include "bleprofile.h"
#include "bleapp.h"
#include "gpiodriver.h"
#include "string.h"
#include "stdio.h"
#include "platform.h"
#include "MyProject.h"
#include "spar_utils.h"
```

6) Now the last step is to create a 'Make Target' bullseye for this project. To do this we simply go over to the 'Make Target' tab and copy one of the existing Make Targets and rename it to our own as shown below…We can use the same hello_sensor make target for this.
Update to SDK2.x.x

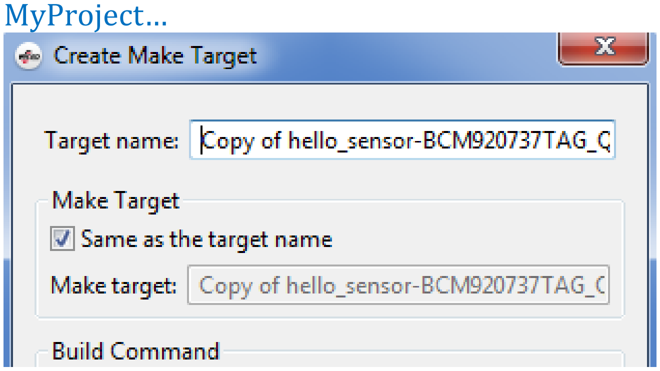


Now go to the top of the Make Target list and right-click and past what we have copied back into the list…

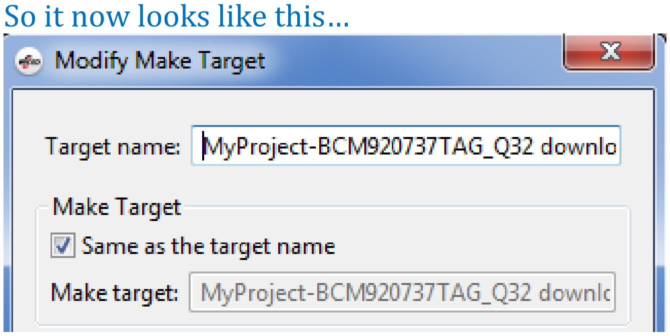




You will now see the dialog indicating that it is a Copy of hello_sensor make target. Just rename it to MyProject…



So it now looks like this…

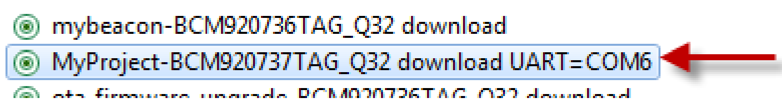


NOTE: You can leave everything else the way it is.

This Make Target finds the files from the folder structure Apps/MyProject . It then calls a make.exe file which is located in the following directory…
C:\'Install Directory'\WICED-Smart-SDK-2.1.0\WICED-Smart-SDK\make.exe

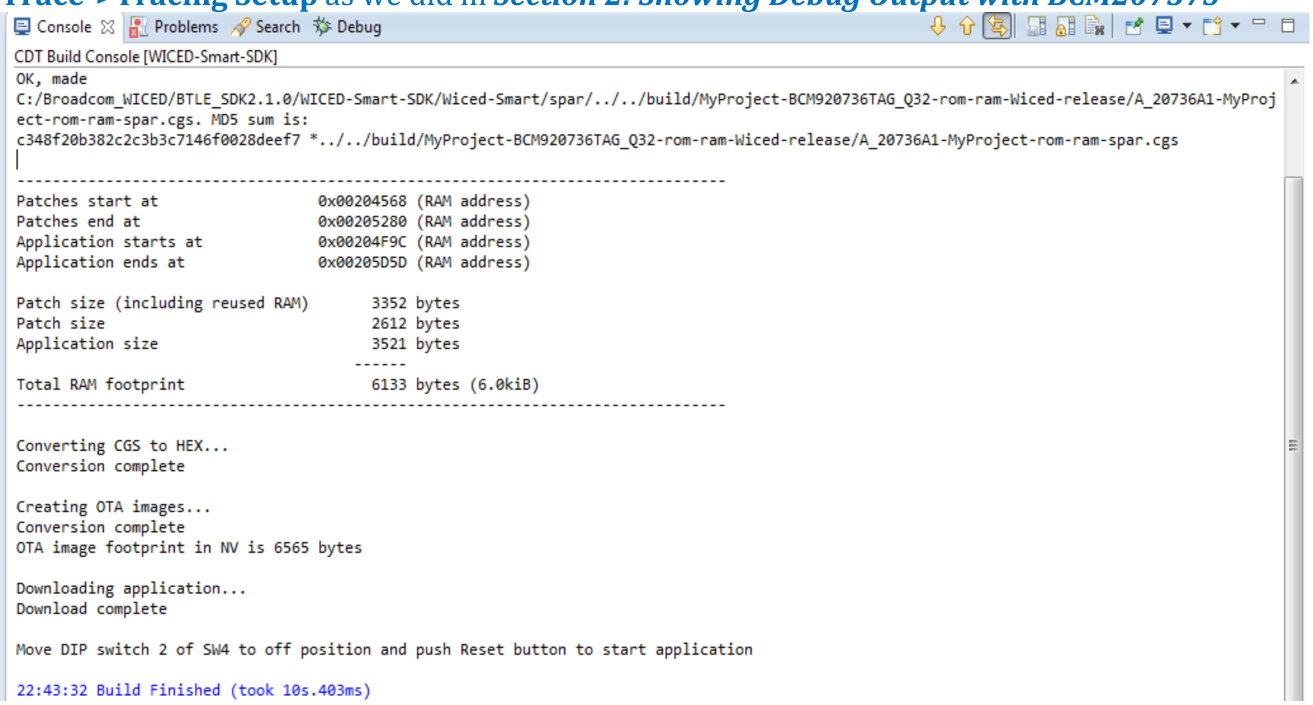
7) Now we can compile and download MyProject to ensure everything is setup correctly. To do so we simply double-click on the MyProject Green Bullseye and we can view the output on the 'Console' Tab at the bottom of Eclipse. If everything is setup correctly and you have the EMRF-20737S-BOB connected to the SDK via a USB-UART you will see the following output in the Console window.

Double-Click:



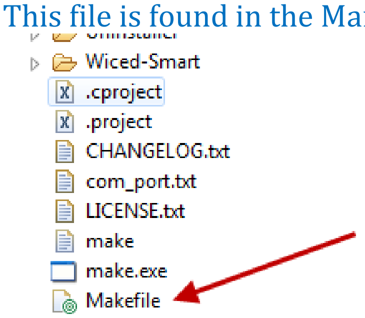
NOTE: I have added the UART=COM6 so that the SDK already knows which COM port to connect to. This is NOT required but makes downloading faster.

Now look at the Console Window you should see the following output…The NOTE about DIP Switch is for the Broadcom TAG board. In our case simply disconnect the HCI_RX line from the USB-UART and hit RST and you will see the Debug Messages in the Console Window by going to **Trace->Start Debug Traces**. If you don't then ensure you configure the correct COM port by selecting **Trace->Tracing Setup** as we did in ***Section 2: Showing Debug Output with BCM20737S***



8) Note that it indicates the application is 'Running' which is true BUT remember if we want to see Debug Messages on a Terminal Window we need to disconnect the HCI_RX line from the FTDI USB-UART TX line so that the HCI_RX line drops low and we press RST on the EMRF-20737S to put the BCM20737S in Application mode. We can now see output on the Terminal Window and we can RESET the device as much as we would like and it will stay in Application Mode. If you want to Re-Program the device just Reconnect the HCI_RX line to the FTDI USB-UART TX line and Hit RST on the EMRF-20737S and the device will be back in Programming Mode.

9) There are times when you may want to have additional output or commands sent to the make.exe executable. There are additional command line options that can be placed in your ***'Make Target'*** that are defined in ***'Makefile'*** located in the same directory as the ***make.exe***. This file is shown below. Contents of ***'Makefile'***. The entire file is not shown but you can view/open it yourself. This file is found in the Main WICED-Smart-SDK Project Directory as shown below.




```

MAKEFILE_TARGETS := help download debug recover clean

include $(SOURCE_ROOT)wiced_toolchain_common.mk


.PHONY: $(MAKEFILE_TARGETS)

BASE_LOCATIONS  := BASErom BASEram BASEflash
SPAR_LOCATIONS  := SPARrom SPARram SPARflash
TOOLCHAINS      := RealView Wiced CodeSourcery
BUILD_TYPE_LIST := debug release

define USAGE_TEXT

Usage: make <target> [download] [debug] [recover] [UART=yyyy] [JOBS=x] [PLATFORM_NV=EEPROM|SFLASH]
       make run

  <target>
    One each of the following mandatory [and optional] components separated by '-'
      * Application (Apps in sub-directories are referenced by subdir.appname)
      * Hardware Platform ($(filter-out common  include README.txt,$(notdir $(wildcard Wiced/Platform/*))))
      * [BASE location] ($(BASE_LOCATIONS))
      * [SPAR location] ($(SPAR_LOCATIONS))
      * [Toolchain] ($(TOOLCHAINS))

  [download]
    Download firmware image to target platform

  [debug]
    Connect to the target platform and run the debugger

  [recover]
    Recover a corrupted target platform

  [VERBOSE=1]
    Shows the commands as they are being executed

  [JOBS=x]
    Sets the maximum number of parallel build threads (default=4)

  [UART=yyyy]
    Use the uart specified here instead of trying to detect the Wiced-Smart device.
    This is useful when you are working on multiple smart devices simultaneously.
```

For example there is there rare scenario when the internal EEPROM of the Device can get corrupted somehow during development and you need to 'Recover' the device. The recovery procedure can be accomplished by the following steps.

1) Force the SDA line HIGH by connecting it via a Jumper wire to either VIN or VREG.
2) While the SDA line is held HIGH Press and Release the RST button.
3) Now you can release the SDA line.
4) This forces the device to boot from ROM and puts it into Programming Mode awaiting programming commands from the SDK.
5) Create a Make Target with the COM Port for the FTDI device you are using and by specifying 'recover' in the Make Target. Your make target will look like this but probably with a different COM Port being assigned.

```
mybeacon-BCM920736TAG_Q32 download
MyProject-BCM920737TAG_Q32 download UART=COM6
MyProject-BCM920737TAG_Q32 recover UART=COM6
```

6) If everything went according to plan you will see the following output on the Console Window.

```
Console  Problems  Search  Debug
CDT Build Console [WICED-Smart-SDK]
OK, made
C:/Broadcom_WICED/BTLE_SDK2.1.0/WICED-Smart-SDK/Wiced-Smart/spar/../../build/MyProject-BCM920736TAG_Q32-rom-ram-Wiced-release/A_20736A1-MyProj
ect-rom-ram-spar.cgs. MD5 sum is:
c348f20b382c2c3b3c7146f0028deef7 *../../build/MyProject-BCM920736TAG_Q32-rom-ram-Wiced-release/A_20736A1-MyProject-rom-ram-spar.cgs

--------------------------------------------------------------------------------
Patches start at                  0x00204568 (RAM address)
Patches end at                    0x00205280 (RAM address)
Application starts at             0x00204F9C (RAM address)
Application ends at               0x00205D5D (RAM address)

Patch size (including reused RAM)      3352 bytes
Patch size                             2612 bytes
Application size                       3521 bytes
                                     ------
Total RAM footprint                    6133 bytes (6.0kiB)
--------------------------------------------------------------------------------

Converting CGS to HEX...
Conversion complete

Creating OTA images...
Conversion complete
OTA image footprint in NV is 6565 bytes

Recovering platform ...
Recovery complete

Move DIP switch 2 of SW4 to off position and push Reset button to start application

23:08:59 Build Finished (took 10s.375ms)
```

NOTE: Another reason you may want to manually specify the COM port is so that you can have multiple EMRF-20737S's plugged into the PC or to speed up the download process so the SDK does not have to search through the COM Ports. This will dictate which one gets programmed.
You have already seen examples of directly specifying the COM port in prior sections.

That wraps it up for a quick overview of How to Create Your Own Project. We will move on to Section 6: Debugging Techniques.

Section 6: Debugging Techniques

Currently the only means to debug the BCM2073x/BCM2073xS devices is to place printf like commands in firmware. The device does have SWD capability but it is currently only supported with a Keil RealView which costs ~$5000. Luckily the BTLE Stack and Peripheral API functions are already done for us so we can still do some pretty cool things without needing a sophisticated debugger. You can use a Segger J-LINK SWD. A link from the forums has been provided below. Keep in mind on these devices you have ~30KB of RAM code space available for your application. This may seem small but you need to keep in mind that the entire BTLE Stack and Peripheral API's are already programmed in ROM. The ~30KB of RAM can truly be set aside for Dynamic Data or your own custom functions as we don't need to spend any of this available space for Peripheral Initialization functions, Peripheral handlers, or BTLE Function calls. It has been indicated that there are some better Debug capabilities coming down the road in future SDK's.

Segger J-LINK WICED Smart Forum Post: How to use a Segger J-LINK.
http://community.broadcom.com/community/wiced-smart/wiced-smart-forums/blog/2014/08/08/wiced-smart-jlink-debugger

So to debug we will make us of the ble_trace commands that have already been created for us. We can search our project for ble_trace by doing CRTL+F and typing ble_trace in the Find/Replace window. It is useful to take a look at the common ones that are used. Some of the common ones are shown below…

To simply print out a Text String we use ble_trace0:

```
    ble_trace0("hello_sensor_create()");
```

To print out a single variable we use ble_trace1:

```
    ble_trace1("NVRAM write:%04x\n", writtenbyte);
```

To print out 2 variables we use ble_trace2:

```
    ble_trace2("hello_sensor_write_handler: bad write len:%d handle:0x%x\n", len, handle);
```

To print out 3 variables we use ble_trace3:

```
    ble_trace3("(INT)But1:%d But2:%d But3:%d\n", value&0x01, (value& 0x02) >> 1,
                                (value & 0x04) >> 2);
```

To print out 4 variables we would use ble_trace4:

```
ble_trace4("EncOn %08x%04x client_configuration:%04x blinks:%d\n",
                (hello_sensor_hostinfo.bdaddr[5] << 24) + (hello_sensor_hostinfo.bdaddr[4] << 16) +
                (hello_sensor_hostinfo.bdaddr[3] << 8) + hello_sensor_hostinfo.bdaddr[2],
                (hello_sensor_hostinfo.bdaddr[1] << 8) + hello_sensor_hostinfo.bdaddr[0],
                hello_sensor_hostinfo.characteristic_client_configuration,
                hello_sensor_hostinfo.number_of_blinks);
```

Hopefully you are starting to see the pattern as to how many variables you want to print out and the ble_traceX option you would use. The ble_traceX functions are defined in ***bleapp.h***.

We now move on to Section 7: How to SLEEP?

Section 7: How to SLEEP?

The BCM2073xS devices have 2 SLEEP Modes. These modes are outlined below.

A. SLEEP

I typically refer to this mode as IDLE. This mode is typically handled automatically by the underlying RTOS in which if there are no Tasks that are active it will automatically place the device into this mode to conserve power.

- Clocks still active
- RAM is retained
- Fast Wake-Up
- Handled automatically by RTOS

B. DEEP SLEEP

This mode is similar to MCU's DEEP SLEEP modes in which they lose the RAM contents and go through a POR/Re-initialization of the system upon waking up. This mode can be used if there will be relatively long interval between sending data or if the device is not in use.

- 128KHz and external 32KHz clocks can be active
- RAM is not retained
- Wake-up require going through POR/Re-Initialization.
- Can be woke up from either a GPIO Interrupt OR a 'Time-Wake' from either the 128KHz clock or the 32kHz clock.
- Lowest Power Mode ~1uA

For this section we will only concern ourselves with DEEP SLEEP and only with using a GPIO Interrupt to wake the device. We will cover the Timed-Wake in either a supplementary Appnote or an addendum to this User Manual at a later date.

Earlier in ***Secton 4.5 hello_sensor create()*** we saw the following lines of code. These lines of code initialize the DEEP SLEEP functionality and indicate that we want to wake from a GPIO source.

The ***devlpm_init()*** initializes the low power mode and the ***devlpm_enableWakeFrom()*** specifies to wake from Deep Sleep from a GPIO interrupt. You can also wake up from a 'timed wake' from either the internal 128kHz LPO or an external 32kHz XTAL. Timed wakeups will be covered in a future Appnote with a code example. ***devlpm_init()*** and ***devlpm_enableWakeFrom()*** are defined in ***devicelpm.c***.

```
// If power save timeout is not enabled, enable device LPM
// If powersave_timeout is enabled, the FW would have enabled it already along with a
// number of other things. This is needed for the app to be able to register a
// callback that is invoked to participate in sleep decisions.
if(!hello_sensor_cfg.powersave_timeout)
{
      ble_trace0("Call devlpm_init and Config GPIO Wakeup\n");
      devlpm_init();
      devlpm_enableWakeFrom(DEV_LPM_WAKE_SOURCE_GPIO);
}
```

Now all we need to do is determine when we would like to put the device into DEEP SLEEP. Often times this may be after a Device has been disconnected from the client/master OR if both Advertising Intervals have 'timed-out' indicating there is nothing around to connect to. Earlier in ***Section 4.7 hello_sensor_connection_down()*** and also in ***Section 4.8 hello_sensor_advertisement_stopped()*** we saw some lines of code shown below. These are the functions to put the device into DEEP SLEEP.

```
else
{
       ble_trace0("Entering DeepSleep - Connection Lost \n");
       bleapputils_delayUs(500);
       bleprofile_PrepareHidOff();
}
```

This else statement followed from an if statement that checks the variable ***hell_sensor_stay_connected*** in which if it is 1 we would typically start the Advertisements again if it is 0 the firmware drops into the else condition and then the ***bleprofile_PrepareHidOff()*** function is executed which puts the device into DEEP SLEEP.

If you use this example you can wake the device by simply pressing P0 push-button or possibly P4 depending on what you have configured in platform.h as the push-button that is configured in the BLTE stack. P0 is the default. Keep in mind you are NOT limited to P0/P4 or what is defined in ***platform.h*** you can configure any GPIO to be an interrupt manually.

That should get you started on understanding how to put the device into its lowest power state, DEEP_SLEEP. As indicated more to follow down the road with 'Timed-Wakeups'.

We now move onto Section 8: Configure GPIO.

Section 8: Configure GPIO

This section is not an 'all-inclusive' section but will provide you at least a basic idea of how to configure GPIO and test the GPIO are all functional on the EMRF-20737S.

One of the first things to point out is that when referring to GPIO Ports P1 does not mean Pin1 there is no correlation to the actual Port # to the Pin #. You need to check the BCM20737S Datasheet for the specific pin numbers and how the correlate to the Port #'s.

NOTES:

- All GPIO can be configured as input, output or disabled(HIGH-Z)
- All GPIO have internal Pull-Ups and Pull-Downs that can be enabled when used as an input
- An output enabled GPIO pin will retain its state in DEEP_SLEEP
- All GPIO's can be configured as edge driven Interrupts(rising/falling/both)
- Since All GPIO's can be configured as an Interrupt they are all capable of waking the system from SLEEP and/or DEEP_SLEEP
- GPIO's can source/sink 2mA. Ports P26, P27, P28 can sink up to 16mA
- Some GPIO's are bonded together and can provide different functionality depending on the Port selection. Only one of the Bonded Ports can be used at a time. The unused pin must be input and output disabled.

A. How GPIO PORTS are Accessed

To properly access Ports we must access them by the correct Port(0-2) and by the correct pin/bit setting in the Port Register. The ports are defined as shown below.

P0 – P15 = PORT0
P16-P31 = PORT1
P32-P38 = PORT2

Or another way to look at this is take the PORT # and divide it by 16. The integer remainder is the PORT #.

To access the correct Port Register Bit we take the PORT# and do a Modulo 16. Using this yields the Following definitions that can be defined. There will be a GPIO_Test.c file that will be available on www.embeddedmasters.com that you can make use of.

```
#define GPIO_P0     0           //Port 0
#define GPIO_P1     1           //EEPROM WP Pin, Be Careful
#define GPIO_P2     2
#define GPIO_P3     3
#define GPIO_P4     4
#define GPIO_P8     8
#define GPIO_P11    11
#define GPIO_P12    12
#define GPIO_P13    13          //Dual Bonded with P28
#define GPIO_P14    14          //Dual Bonded with P38
#define GPIO_P15    15          //Port 0
```

```
#define GPIO_P24     8            //Port 1
#define GPIO_P25     9            //Port 1
#define GPIO_P26    10            //Port 1
#define GPIO_P27    11            //Port 1
#define GPIO_P28    12            //Port 1     //Dual Bonded with P13


#define GPIO_P32     0            //Port 2
#define GPIO_P33     1            //Port 2
#define GPIO_P38     6            //Port 2     //Dual Bonded with P14
```

It may also make sense to create a more meaningful name for the individual ports as shown below.

```
#define GPIO_PORT0   0
#define GPIO_PORT1   1
#define GPIO_PORT2   2
```

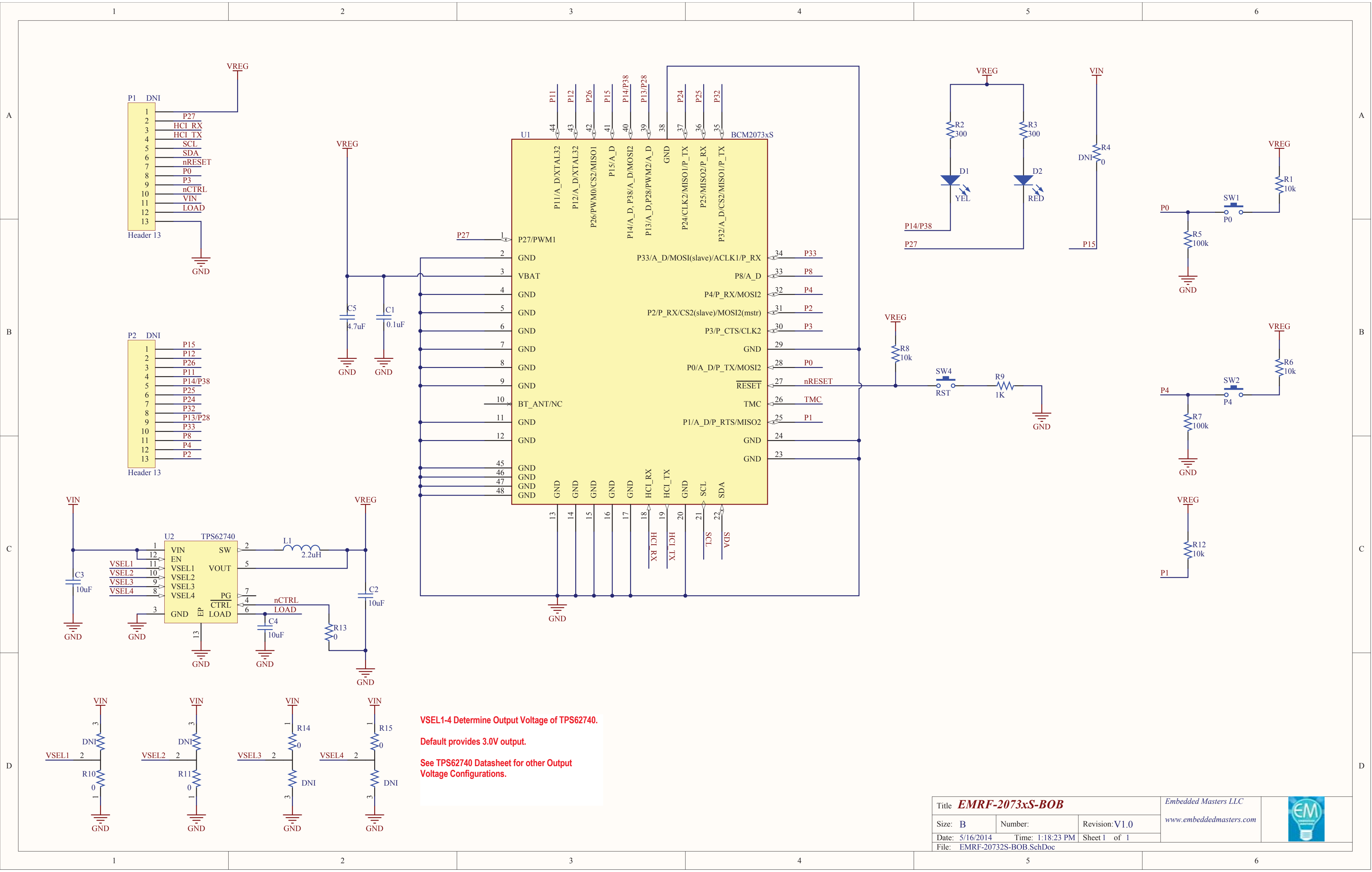
So now using these #defines we can easily make use of a GPIO function. All GPIO functions are defined in the ***'Drivers'*** folder in the SDK and in particular in ***gpiodriver.h***.

```
gpio_configurePin(GPIO_PORT0, GPIO_P0, GPIO_OUTPUT_ENABLE, GPIO_HIGH);

gpio_configurePin(GPIO_PORT0, GPIO_P1, GPIO_OUTPUT_ENABLE, GPIO_HIGH);

gpio_configurePin(GPIO_PORT0, GPIO_P2, GPIO_OUTPUT_ENABLE, GPIO_HIGH);

gpio_configurePin(GPIO_PORT0, GPIO_P3, GPIO_OUTPUT_ENABLE, GPIO_HIGH);

gpio_configurePin(GPIO_PORT0, GPIO_P4, GPIO_OUTPUT_ENABLE, GPIO_HIGH);
```

Again these are not by any means all inclusive of the GPIO functions but just one example of how you would go about accessing the correct Port# and Port Register bit for an individual Port and then to show how these would be used with the ***gpio_configurePin()*** function. There will be more examples to follow in future Addendums to this User Manual and/or Appnotes showing more functionality.

Revision History:

V1.0.1: Added Silkscreen errata. Fixed a couple of formatting errors.



P1 DNI
Header 13
1 P27
2 HCI_RX
3 HCI_TX
4 SCL
5 SDA
6 nRESET
7 P0
8 P3
9 nCTRL
10 VIN
11 LOAD
12
13
P2 DNI
Header 13
1 P15
2 P12
3 P26
4 P11
5 P14/P38
6 P25
7 P24
8 P32
9 P13/P28
10 P33
11 P8
12 P4
13 P2
VREG
GND
VIN
U1
BCM2073xS
P27/PWM1
VBAT
BT_ANT/NC
P11/A_D/XTAL32
P12/A_D/XTAL32
P26/PWM0/CS2/MISO1
P15/A_D
P14/A_D_P38/A_D/MOSI2
P13/A_D_P28/PWM2/A_D
P24/CLK2/MISO1/P_TX
P25/MISO2/P_RX
P32/A_D/CS2/MISO1/P_TX
P33/A_D/MOSI(slave)/ACLK1/P_RX
P8/A_D
P4/P_RX/MOSI2
P2/P_RX/CS2(slave)/MOSI2(mstr)
P3/P_CTS/CLK2
P0/A_D/P_TX/MOSI2
RESET
TMC
P1/A_D/P_RTS/MISO2
HCI_RX
HCI_TX
SCL
SDA
nRESET
C5 4.7uF
C1 0.1uF
R2 300
R3 300
D1 YEL
D2 RED
R4 DNI 0
R1 10k
R5 100k
SW1 P0
R8 10k
SW4 RST
R9 1K
R6 10k
R7 100k
SW2 P4
R12 10k
U2 TPS62740
VIN
EN
VSEL1
VSEL2
VSEL3
VSEL4
GND
EP
SW
VOUT
PG
CTRL
LOAD
L1 2.2uH
C3 10uF
C2 10uF
C4 10uF
R13 0
nCTRL
LOAD
VSEL1
VSEL2
VSEL3
VSEL4
DNI
R10 0
R11 0
R14 0
R15 0
VSEL1-4 Determine Output Voltage of TPS62740.
Default provides 3.0V output.
See TPS62740 Datasheet for other Output Voltage Configurations.
Title EMRF-2073xS-BOB
Size: B
Number:
Revision: V1.0
Date: 5/16/2014
Time: 1:18:23 PM
Sheet 1 of 1
File: EMRF-20732S-BOB.SchDoc
Embedded Masters LLC
www.embeddedmasters.com